UNIVERSIDAD DE LA RIOJA

TRABAJO FIN DE ESTUDIOS

Título

Integración de JupyterHub con Kubernetes para la ejecución de cargas de trabajo de Data Science en contenedores Docker

Autor/es

LUIS CABEZÓN MANCHADO

Director/es

CÉSAR DOMÍNGUEZ PÉREZ

Facultad

Facultad de Ciencia y Tecnología

Titulación

Grado en Ingeniería Informática

Departamento

MATEMÁTICAS Y COMPUTACIÓN

Curso académico

2019-20

publicaciones.unirioja.es
E-mail: publicaciones@unirioja.es

UNIVERSIDAD DE LA RIOJA

Facultad de Ciencia y Tecnología

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Integración de JupyterHub con Kubernetes para la ejecución de cargas de trabajo de Data Science en contenedores Docker

Realizado por:

Luis Cabezón Manchado

Tutelado por:

César Domínguez Pérez

Carlos Acedo Nieto

Logroño, junio, 2020

Índice

Resumen

Durante los últimos años se está produciendo una migración de los servidores privados de las compañías hacia nubes públicas como Amazon o Google. En cierta medida, esto se debe a la creciente popularidad de tecnologías como Docker, permitiendo aislar los diferentes servicios que corren dentro del mismo servidor. La expansión de Docker genera una necesidad de orquestar dichos contenedores. Estos orquestadores permiten la gestión, control y despliegue de servicios dentro de una red multinodo. El orquestador más utilizado es Kubernetes, tecnología creada por Google y liberada más tarde.

Uno de los principales problemas a la hora de migrar centros orientados a la producción de modelos estadísticos es la carencia de flexibilidad que se le da al usuario para personalizar su entorno de trabajo. Jupyter Enterprise Gateway nace con la idea de mitigar dicho problema, permitiendo a los usuarios ejecutar cargas en un entorno multinodo haciendo uso de librerías preinstaladas en una imagen de Docker. Estas imágenes son estáticas, teniendo que instalar las librerías cada vez que se ejecute de nuevo la imagen.

La idea de este proyecto surge como la necesidad de aumentar la flexibilidad de Jupyter Enterprise Gateway. Esta mejora proporcionará al usuario el control de sus propios entornos, permitiendo la instalación de librerías en entornos de forma dinámica, aliviando la problemática de tener que instalarlas cada vez que se usa la imagen.

Asimismo, se utilizará la tecnología Terraform con el fin de evitar sobrecostes durante los períodos de tiempo de inactividad de la plataforma. Dicha tecnología permite automatizar el despliegue de plataformas en diferentes nubes públicas, permitiendo que ésta sea portable y configurable.

Abstract

In recent years, there has been a migration of companies' private servers to public clouds like Amazon or Google. To a certain extent, this is due to the increasing popularity of technologies such as Docker, which could isolate the different services that run within the same server. The Docker expansion creates a need to orchestrate these containers. These orchestrators allow the management, control and deployment of services within a multinode network. The most widely used orchestrator is Kubernetes, a technology created by Google and released later.

One of the main problems when migrating centers oriented to the production of statistical models is the flexibility that is given to the user to personalize their work environment. Jupyter Enterprise Gateway was born with the idea of mitigating this problem, to users running loads in a multinode environment using pre-installed libraries in a Docker image. These images are static, having to install the libraries every time the new image is run.

The idea of this project arises as the need to increase the flexibility of the Jupyter Enterprise Gateway. This improvement will provide the user with control of their own environments, installing libraries in environments dynamically, alleviating the problem of having to install them every time the image is used.

In addition, Terraform technology will be used in order to avoid cost overruns during platform downtime. This technology allows to automate the deployment of platforms in different public clouds, locating it to be portable and configurable.

Prólogo

Por Carlos Acedo Nieto (SDG Group)

Los Notebooks Interactivos (o simplemente Notebooks) son la herramienta más extendida entre los Data Scientists para soportar su flujo de trabajo habitual, en el que tanto la experimentación como las anotaciones son parte fundamental. Se trata de una interfaz de desarrollo donde bloques de código ejecutables, resultados y notas de contexto se van sucediendo, en lo que podría considerarse como una suerte de cuaderno digital que recoge todos los pasos de un experimento o ejercicio. De entre las herramientas de Notebooks disponibles, se puede afirmar que Jupyter es el estándar *de facto* en torno al desarrollo de modelos de Machine Learning con Python en todo el mundo[1].

Jupyter Notebook nace como una herramienta de trabajo individual (monousuario), pero a medida que la Analítica Avanzada se va consolidando en la industria y se revelan nuevas necesidades propias del software en el entorno empresarial, surgen nuevas iniciativas bajo el paraguas de Jupyter para darles respuesta. Tal es el caso de Jupyter Hub, un servicio que proporciona capacidades multiusuario al permitir lanzar múltiples instancias de Jupyter Notebook en el servidor donde se ejecuta; o Jupyter Enterprise Gateway, que proporciona escalabilidad horizontal permitiendo que las ejecuciones de código de los Notebooks se lleven a cabo en un servidor remoto (o clúster) en lugar de en la máquina en la que se ejecuta Jupyter Notebook.

Jupyter Enterprise Gateway abre la puerta al escalado sin límite de cargas de trabajo de Analítica Avanzada, una cuestión indispensable en entornos corporativos. No obstante, presenta algunas limitaciones derivadas de su inmadurez que está previsto solventar según la hoja de ruta del proyecto, aunque por el momento no hay fechas de lanzamiento de nuevas características. Esto, unido a la necesidad detectada en el mercado por SDG (en concreto en un cliente del sector de la banca), motivó que se propusiera al autor desarrollar su TFG en torno a la implementación de algunas de las características más interesantes de la hoja de ruta de Jupyter Enterprise Gateway, en concreto las denominadas como *User Environments* y *Kernel Configuration Profile*.

Cabe destacar la trascendencia del presente TFG dada la dimensión del proyecto Jupyter[2] en el mundo y de Jupyter Enterprise Gateway a nivel corporativo. A día de hoy la implementación de estas características enriquece el porfolio de SDG en materia de Plataformas para Analítica Avanzada, e incluso podría llegar a suponer un factor diferencial con respecto a sus competidores en algunos casos. Además, se ha iniciado el trámite de integración de algunas partes del código fuente en el repositorio de código oficial de Jupyter Enterprise Gateway[3], por lo que muchas corporaciones de todo el mundo podrán beneficiarse del excelente trabajo que el autor ha realizado en el desarrollo de este TFG.

[1] El proyecto Jupyter cuenta dos millones de usuarios en todo el mundo. Ver https://www.linkedin.com/company/project-jupyter/about/

[2] Jupyter está financiado directamente por compañías como Google, Microsoft o la Comisión Europea, y soportado por socios institucionales como Netflix, Bloomberg o Amazon Web Services. Ver https://jupyter.org/about

[3] La integración de la totalidad del código fuente (y por tanto de la totalidad de las características implementadas) en el repositorio oficial está supeditada a la adaptación del código a estándares de desarrollo particulares del proyecto Jupyter que se llevarán a cabo con posterioridad.

1 – Planificación

Durante esta sección se detallan tanto aspectos que han propiciado la realización del proyecto como aspectos necesarios para la organización de este.

1.1 – Motivación

Actualmente los entornos de desarrollo online alrededor de tecnologías como Python o R están tomando una gran relevancia en el mercado. Esto se debe en parte a que permiten centralizar los desarrollos desde el punto de vista de la seguridad, el acceso al dato y los recursos.

El problema de dichos entornos de desarrollo es la dificultad a la hora de escalar dicha arquitectura en una infraestructura multinodo. En entornos JupyterHub, como en nuestro caso, esta problemática está parcialmente solventada por componentes como Jupyter Enterprise Gateway. Este componente carece de la flexibilidad necesaria en términos de gestión de dependencias entre paquetes desde el punto de vista del usuario.

1.2 – Alcance

La idea es reproducir en local, y más tarde en la nube de Amazon, una arquitectura basada en la escalabilidad horizontal y gestión óptima de recursos hardware (memoria RAM, CPU, GPU…) para la ejecución de cargas de trabajo de Data Science.

Esta arquitectura estará basada en Docker, tecnología que proporciona una capa de abstracción y automatización de virtualización de aplicaciones. El sistema estará compuesto por un clúster de Kubernetes, tecnología que permite la gestión de recursos entre distintos nodos. Dicho clúster estará formado por un grupo de nodos autogestionado, permitiendo que el servicio de Amazon gestione el número de nodos en función de las necesidades.

El objetivo es lanzar las cargas contra el clúster de Kubernetes desde un notebook de Jupyter, de forma que se usen los recursos del clúster y no del sistema donde se ejecuta el notebook. Cuando el usuario inicie sesión en JupyterHub, el sistema filtrará los kernels en función del usuario autenticado, pudiendo ver éste únicamente sus entornos.

Para ello, se modificará el paquete Jupyter Enterprise Gateway para aumentar la flexibilidad a la hora de permitir al usuario la gestión de dependencias entre paquetes.

Por último, se automatizará el proceso de despliegue de la plataforma para facilitar la portabilidad de esta.

1.3 – Metodología

Tras analizar las distintas metodologías existentes para proyectos orientados al mercado de las TIC, se ha decidido utilizar la metodología iterativo-incremental. El motivo principal es el desconocimiento de las limitaciones que vamos a encontrar al hacer uso de las tecnologías explicadas en la siguiente sección. De esta forma, se podrá llevar un seguimiento más real del proyecto, permitiendo cambios en las especificaciones iniciales dictadas por el cliente.

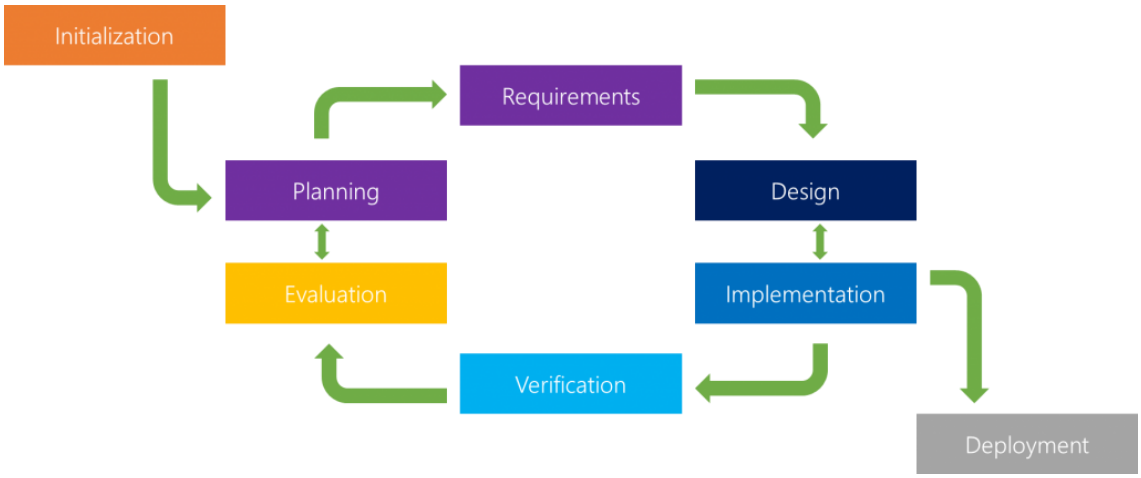


Ilustración 1 - Metodología iterativo e incremental

Se partirá de una situación base donde se probará la integración entre los diferentes elementos que se usarán a lo largo del proyecto.

El proyecto se divide en incrementos (ver Ilustración 1), los cuales pasan por las siguientes fases:

1. **Análisis**: Fase donde se establecen los nuevos requisitos, funcionales y no funcionales, que se deben añadir durante este incremento.
2. **Diseño**: Fase donde se plantea si el diseño actual permite alcanzar los nuevos objetivos propuesto para dicho incremento.
3. **Implementación**: Fase durante la cual se implementan/modifican las funciones del código.
4. **Despliegue:** Fase durante la cual se despliega la aplicación en un entorno local.
5. **Verificación:** Se realizan las pruebas pertinentes tratando de comprobar el correcto funcionamiento de los nuevos cambios.
6. **Evaluación**: Evaluar tanto el trabajo hecho evaluando el proceso (carga de trabajo, problemas encontrados…)

1.4 – EDT: Estructura de descomposición del trabajo

En esta sección se presenta la estructura de descomposición del trabajo, o EDT, para el proyecto. En siguiente ilustración vemos reflejado los distintos paquetes de trabajo que formar dicha EDT.

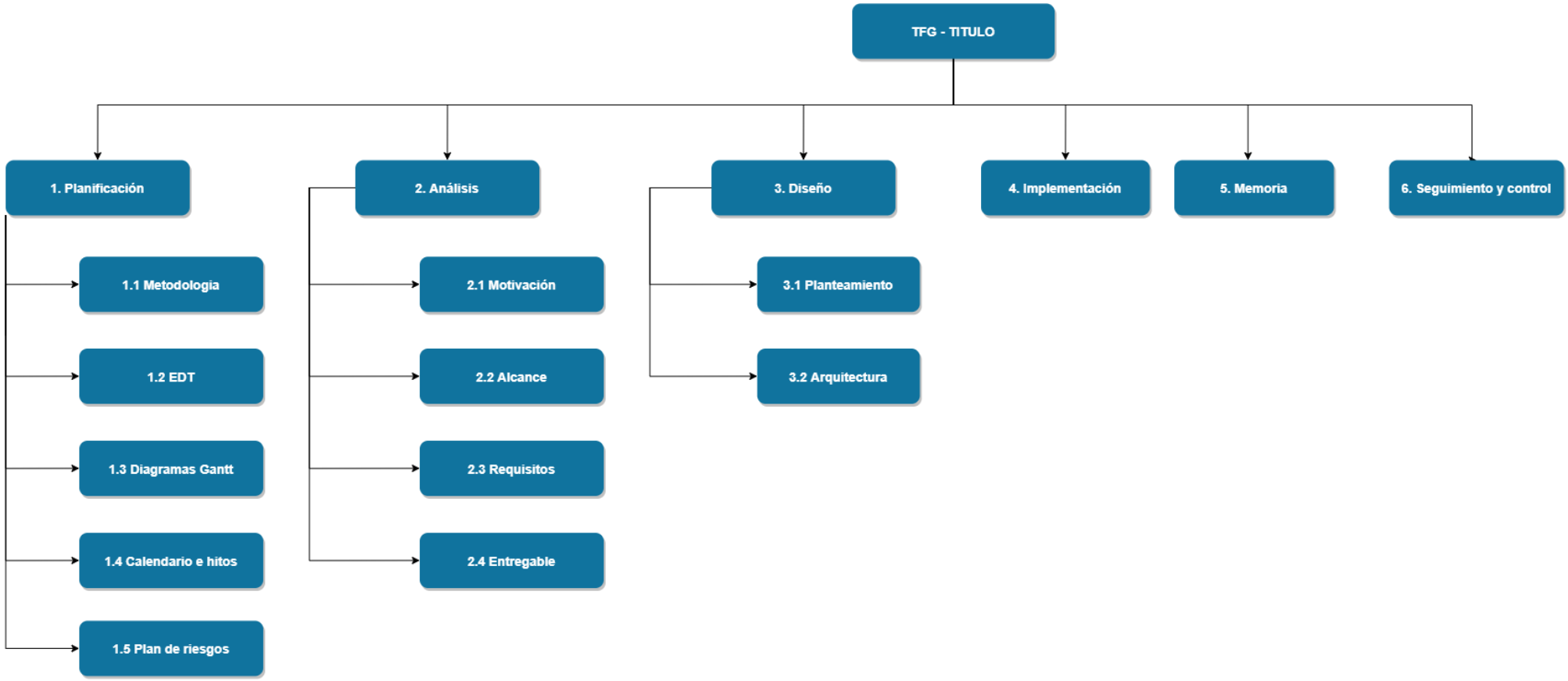


Ilustración 2 - EDT del proyecto

La siguiente tabla expone cada uno de los paquetes de trabajo que forman la EDT junto a una breve descripción de ellos.

ID	Nombre	Descripción
1	**Planificación**	
1.1	Metodología	Elección de la metodología a seguir durante la realización del proyecto.
1.2	EDT	Estructura de descomposición del trabajo para la realización del proyecto.
1.3	Diagramas Gantt	Planificación y estimación de la duración de cada tarea a lo largo del proyecto.
1.4	Calendario e hitos	Estructura de las semanas durante el segundo semestre
1.5	Plan de riesgos	Plan de actuación ante riegos posibles a lo largo del proyecto.
2	**Análisis**	
2.1	Motivación	Causas que han propiciado la realización del proyecto.
2.2	Alcance	Definición del alcance del proyecto.
2.3	Requisitos	Requisitos funcionales y no funcionales iniciales del proyecto.
2.4	Entregables	Entregables generados durante el proyecto.
3	**Diseño**	
3.1	Planteamiento	Objetivo principal y punto de partida
3.2	Tecnologías	Tecnologías usadas durante la realización del proyecto.
3.3	Arquitectura	Definición de la arquitectura a implementar.
3.4	Solución final	Definición de la solución integrando el paquete Jupyter Enterprise Gateway en la arquitectura.
4	**Implementación**	
4.1	Clúster Kubernetes local	Creación de la arquitectura en local.
4.2	Modificación de paquetes Jupyter	Modificación del paquete Jupyter Enterprise Gateway y Jupyter Notebook para implementar los requisitos.
4.3	Clúster Kubernetes en la nube	Creación de la arquitectura en la nube de Amazon.
4.4	Automatizar despliegue	Automatización del despliegue de la arquitectura en AWS haciendo uso de la tecnología Terraform
5	**Memoria**	
6	**Seguimiento y control**	

Tabla 1 - Descripción de los paquetes de trabajo

1.5 – Diagramas Gantt

Tarea	estimadas	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	S13	S14	S15	S16
	300																
Planificación	**20**																
Metodología																	
EDT																	
Diagramas Gantt																	
Calendario e hitos																	
Plan de riesgos																	
Análisis	**25**																
Motivación	3																
Alcance	7																
Requisitos	10																
Entregables	5																
Diseño	**50**																
Planteamiento	5																
Tecnologías	5																
Arquitectura	15																
Solución final	25																
Implementación	**170**																
Clúster Kubernetes en local	25																
Modificación de paquetes Jupyter	70																
Clúster Kubernetes en la nube	35																
Automatizar despliegue	20																
Memoria	**15**																
Seguimiento y control	**20**																

Tabla 2 - Planificación temporal del proyecto

1.6 – Calendario e hitos

En este apartado se muestra el calendario de la universidad de La Rioja durante el curso 19/20. El trabajo de fin de grado ha sido realizado durante el segundo cuatrimestre (3 de febrero – 22 de mayo)

Tabla 3 - Calendario Universidad de La Rioja 19/20

1.7 – Plan de Riesgos

En esta sección definiremos la gestión de las comunicaciones con los responsables del proyecto en el ámbito empresarial y académico.

Además, se incluyen diferentes riesgos a los que puede estar expuesto el proyecto a lo largo de su tiempo de vida. Para cada uno de ellos se expone causa/motivo, nivel de impacto y la forma de actuar ante el mismo.

1.7.1 – Gestión de comunicaciones

En esta sección hay que diferenciar la comunicación entre la empresa y el tutor académico de la universidad.

Comunicación con el tutor académico

Tipo de comunicación	Informar	Pedir información	Llegar a acuerdo
Síncrona	Reunión cada dos semanas con duración inferior a 1 hora.	Reunión cada dos semanas con duración inferior a 1 hora.	Reunión cada dos semanas con duración inferior a 1 hora.
Asíncrona	Correo electrónico: Outlook	Correo electrónico: Outlook	Correo electrónico: Outlook

Tabla 4 - Gestión de comunicaciones con el tutor

Comunicación dentro del proyecto empresarial

Tipo de comunicación	Informar	Pedir información	Llegar a acuerdo
Síncrona	Reunión todos los viernes con duración de 1 hora	Reunión todos los viernes con duración de 1 hora	Reunión todos los viernes con duración de 1 hora
Asíncrona	-Correo electrónico: Gmail -Hangouts	-Correo electrónico: Gmail -Hangouts	-Correo electrónico: Gmail -Hangouts

Tabla 5 - Gestión de la comunicación en la empresa

1.7.2 – Plan de Riesgos

Las siguientes tablas muestran los riesgos inherentes a un proyecto de larga duración clasificados en función del actor. Para cada uno de ellos se expone la estrategia a seguir para paliar al máximo los impactos negativos en el proyecto.

Riesgo	Estrategia preventiva	Estrategia de minimización	Estrategia de contingencia
Ausencia del tutor académico	No se puede prevenir	Mantener la planificación y seguirla de manera autónoma.	Trata de mantener la comunicación asíncrona, y en el caso extremo apoyarse en otros tutores.
Ausencia del tutor de la empresa	No se puede prevenir	Mantener la planificación y seguirla de manera autónoma.	Trata de mantener la comunicación asíncrona, y en el caso extremo apoyarse en otros tutores.

Tabla 6 - Plan de riesgos asociado a tutores

Riesgo	Estrategia preventiva	Estrategia de minimización	Estrategia de contingencia
Incompatibilidad entre tecnologías	Estudiar el funcionamiento de cada tecnología por separado y de forma conjunta	Tratar de modificar alguna tecnología para que realice nuestras necesidades	Reformular que tecnologías deben estar y cuáles no
Dificultad en el uso	Estudiar el funcionamiento de la tecnología	Búsqueda de dudas en foros como GitHub.	

Tabla 7 - Plan de riesgos asociado a la tecnología

Riesgo	Estrategia preventiva	Estrategia de minimización	Estrategia de contingencia
Número de horas reales mayor a las planificadas	Realizar una buena planificación limitando el alcance del proyecto.	Dedicarse primero a los requisitos de mayor peso.	Reunirse con la empresa y decidir qué requisitos pueden eliminarse y cuáles no.
Número de horas reales menor a las planificadas	Realizar una buena planificación	Desgranar los requisitos en un mayor número de estos.	Reunirse con el tutor/empresa y ver que funcionalidad añadir extra.
Pérdida de documentación/código	Usar OneDrive		

Tabla 8 - Plan de riesgos asociado a las horas

2 - Análisis

Durante esta sección se expondrán los requisitos, tanto funcionales como no funcionales, del proyecto, una breve introducción a las tecnologías principales que van a ser usadas, los entregables que se facilitarán en el momento de la entrega del proyecto y, por último, un estudio del estado del arte de la situación actual.

2.1 – Requisitos

En esta sección se presentan los requisitos tanto funcionales como no funcionales de este proyecto.

2.1.1 - Requisitos funcionales

Cód. Req Fun	Descripción
1	Gestión de entornos conda
2	Recursos gestionados por un orquestador
3	Permitir el filtrado por usuario en Jupyter Enterprise Gateway
4	Gestión de recursos en función del proyecto
5	Automatizar el despliegue en AWS

Tabla 9 - Requisitos funcionales

Requisito Funcional Nº 1 – Gestión de entornos conda

Por defecto Jupyter Enterprise Gateway no permite la gestión de entornos conda, ofreciendo únicamente una serie de entornos predefinidos.

El objetivo es brindar al usuario la posibilidad de crear sus propios entornos personalizados.

Requisito Funcional Nº 2 - Recursos gestionados por un orquestador

El objetivo es separar el servidor de JupyterHub del lugar donde se ejecutan las cargas. Dichas cargas se distribuirán entre una serie de nodos en función del nivel de carga de estos.

Requisito Funcional Nº 3 - Permitir el filtrado por usuario en Jupyter Enterprise Gateway

Jupyter Enterprise Gateway no permite parámetros de petición. Todos los usuarios tienen los mismos entornos predefinidos.

El objetivo es permitir el envío de parámetros de petición con el fin de dotar de una cierta lógica de filtrado a Jupyter Enterprise Gateway.

Requisito Funcional Nº 4 – Gestión de recursos en función del proyecto

Establecer una lógica que permita fijar unos límites a un kernel en función del proyecto al que está relacionado.

Requisito Funcional Nº 5 – Automatizar el despliegue en AWS

Crear una plataforma en AWS portable y desplegable de forma automática usando Terraform.

2.1.2 - Requisitos no funcionales

Cód. Req No Fun	Descripción
1	Añadir seguridad extra a JupyterHub
2	Limitar los puertos abiertos en la máquina de Jupyterhub
3	Uso de herramientas para el control de versiones

Tabla 10 - Requisitos no funcionales

Requisito no funcional N.º 1 – Añadir seguridad extra a JupyterHub

Utilizar el servicio de AWS Cognito para poder gestionar a los usuarios de forma externa a la máquina de JupyterHub

Requisito no funcional N.º 2 - Limitar los puertos abiertos en la máquina de Jupyterhub

Limitar los puertos de la máquina Jupyterhub a estrictamente los necesarios. Entre ellos estarán el HTTPS y NFS. Durante el desarrollo estará abierto el puerto 22 (SSH) para pruebas.

Requisito no funcional N.º 3 – Uso de herramientas para el control de versiones

Gestión de las versiones de código a través de Bitbucket con el fin de poder recuperar versiones antiguas y entender el proceso hasta llegar a la última versión.

2.2 – Tecnologías empleadas durante la realización del TFG

Durante la siguiente sección se explican cada una de las tecnologías más importantes que van a ser usadas a lo largo del proyecto.

2.2.1– JupyterHub

JupyterHub es una herramienta *open Source* desarrollada por Jupyter. Permite liberar al usuario de la parte de instalación y mantenimiento del sistema, siendo accesible como si de un servicio web se tratara, como podemos ver en la Ilustración 3.

JupyterHub puede ejecutar sus entornos de trabajo, también conocidos como notebook, de manera remota o local. En nuestro caso, los notebooks se ejecutarán de forma remota haciendo uso del paquete de utilidades *jupyter-enterprise-gateway.* De esta forma, podremos lanzar las cargas a un servidor remoto, liberándonos de la necesidad de tener un equipo potente.

Las principales características de JupyterHub son:

- **Personalizable**: puede ejecutar código de diferentes lenguajes de programación como R o Python. Respecto a este último, podemos usar nuestros propios entornos conda. Conda es un sistema de gestión de paquetes y entornos, permitiendo aislar las dependencias entre paquetes a nivel de entornos.
- **Flexible:** brinda una capa de seguridad al acceso de los jupyter-notebooks. Los usuarios pueden iniciar sesión contra numerosos protocolos de autenticación (PAM, GitHub).
- **Portable:** gracias a ser un proyecto de código abierto y estar diseñado para ejecutarse en diferentes arquitecturas. Estas arquitecturas pueden ser: entornos cloud, máquinas virtuales o en nuestro propio ordenador.

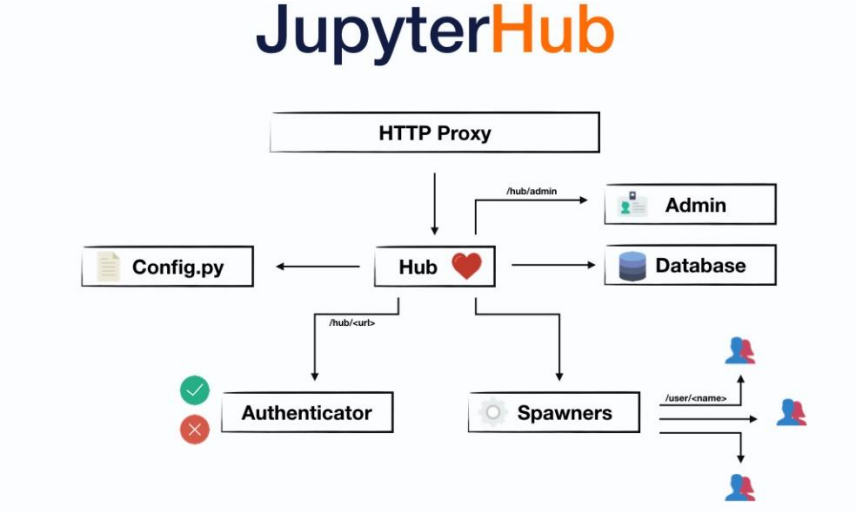


Ilustración 3 - Arquitectura JupyterHub

Una de las principales ventajas de JupyterHub es el uso de extensiones que permiten aumentar la experiencia de usuario. Entre ellas, una de las más relevantes de cara al desarrollo de este proyecto es *nb_conda_kernels*.

Esta extensión permite al usuario lanzar jupyter notebooks con sus propios entornos conda. Conda es un gestor de paquetes y permite la creación de entornos. Cada entorno contiene diferentes paquetes que están aislados del resto de entornos. De esta forma, se resuelven incompatibilidades entre versiones de diferentes paquetes.

2.2.2- Jupyter Enterprise Gateway

Este paquete de utilidades es el punto de unión entre los jupyter-notebook (ejecutados de forma local sobre una máquina) y el clúster de Kubernetes. Una forma intuitiva de entender el funcionamiento es la ilustración 4, donde se representa la separación entre el equipo que corre Jupyter y el clúster de Kubernetes (donde se ejecutan las cargas)

Jupyter Enterprise Gateway, de ahora en adelante JEG, permite la gestión de kernels de forma remota en un entorno clúster. La funcionalidad principal es la gestión de recursos hardware y su optimización, permitiendo que cada kernel use únicamente el hardware que requiere.

Desde una perspectiva técnica, JEG funciona como una API capaz de gestionar kernels en notebooks remotos. De esta forma, cada celda de código de un notebook de Jupyter no se ejecutará en local usando los recursos del sistema anfitrión, sino que se ejecutarán de forma remota en un clúster de Kubernetes.

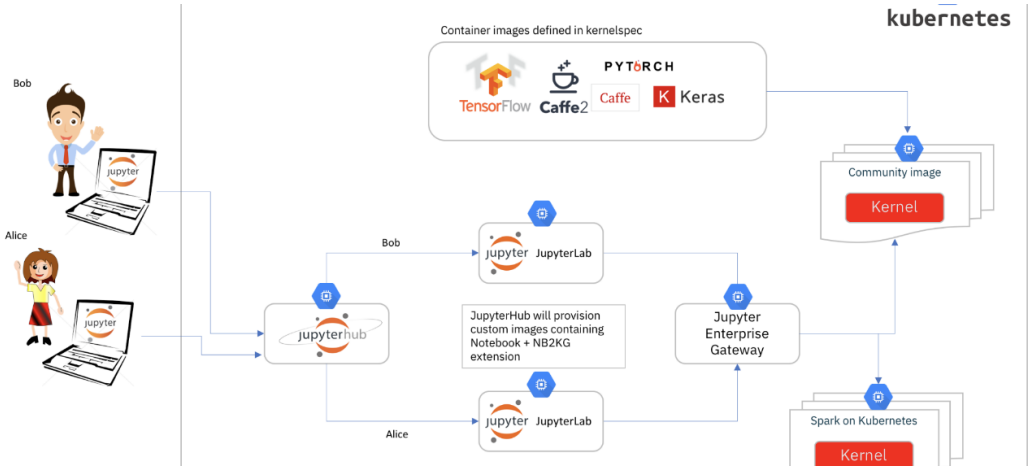


Ilustración 4 - Arquitectura Jupyter Enterprise Gateway

Por defecto, Jupyter Enterprise Gateway carece de la gestión de entornos a nivel de usuario. Esto limita el abanico de posibilidades de uso en un entorno de producción ya que para cada entorno se necesitaría crear una imagen Docker con las librerías necesarias. El problema surge a la hora de instalar nuevas librerías, ya que sería necesario volver a crear esa imagen Docker.

2.2.3 – Docker

La idea principal de Docker es crear contenedores (aplicaciones) ligeros y portables que funcionen en cualquier máquina con Docker instalado, independientemente del sistema operativo instalado en el sistema anfitrión.

Docker se gestiona a través de una API que puede ser llamada desde la herramienta propia de Docker en línea de comandos, desde librerías creadas *exprofeso* o realizando peticiones al endpoint del servicio Web que proporciona dicha API.

Cada contenedor permite empaquetar una aplicación con todas las herramientas necesarias, como librerías o ficheros, dentro de un paquete. Una vez empaquetado, al ejecutarlo en cualquier máquina se obtendrá el mismo resultado.

Ilustración 5 - Proceso de creación de imágenes Docker

El resultado de empaquetar la aplicación es una imagen Docker. Este fichero contiene las directivas necesarias para lanzar la aplicación en un estado determinado (siempre será el mismo). Para construir una imagen será necesario un fichero, llamado Dockerfile, que contenga las directrices necesarias. Cada una de estas imágenes pueden ser ejecutadas en diferentes equipos sin importar el sistema operativo instalado ni las librerías instaladas (Ilustración 5)

Estas imágenes Docker las podemos subir a un repositorio público como DockerHub. Cualquier usuario puede crear sus imágenes Docker y subirlas (ver Ilustración 6). De esta forma, podremos referenciar dichas imágenes en los contenedores sin necesidad de tenerlas descargadas en local.

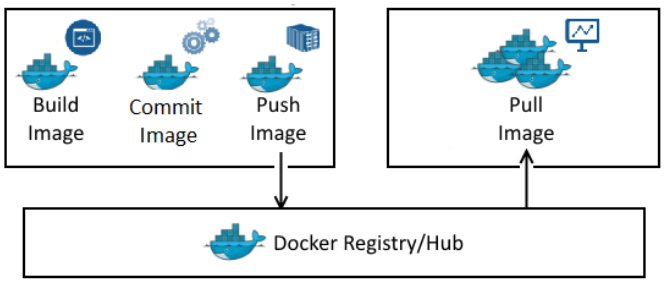


Ilustración 6 - Comandos para publicar/descargar imágenes

2.2.4 – Kubernetes

Kubernetes, también llamada K8s, es un sistema de orquestación de aplicaciones ejecutadas dentro de contenedores Docker. La arquitectura básica de cada uno de los nodos del clúster está representada en la Ilustración 7.

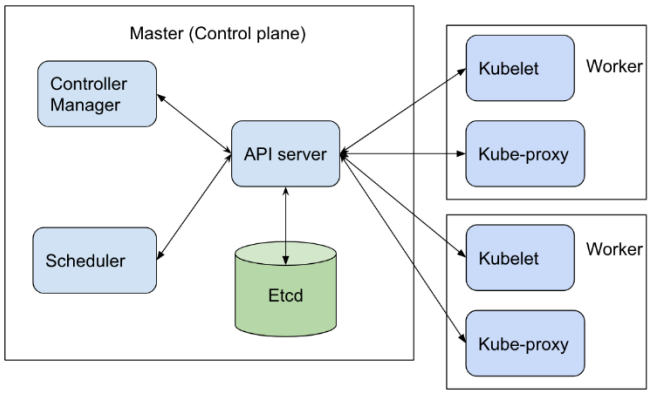


Ilustración 7 - Estructura básica de un clúster de K8s

La orquestación de contenedores consiste en la creación automática de contenedores, así como de su escalado, en función de los recursos asignados a cada contenedor y su fichero de configuración.

La unidad básica dentro de Kubernetes es el POD, que es un conjunto de uno o varios contenedores. Cada o POD se encuentran alojados dentro del misma máquina física y comparten recursos.

Cada POD tiene una dirección de IP y puerto compartidos por todos los contenedores pertenecientes al mismo. La comunicación entre contenedores puede producirse a través de localhost, semáforos y/o memoria compartida. Las aplicaciones que corren dentro de un POD tienen que ser tratadas de forma efímera, ya que no son capaces de resistir a diferentes fallos. Es el orquestador quién se encarga de mantener el estado de un POD, replicándolo cuando sea necesario.

Por otro lado, existe la figura del *Controller Manager.* Este controlador vigila el estado del cluster a través de la API y realiza los cambios necesarios para llegar al estado deseado.

Las funciones más importantes del controlador son:

- **Control de nodo**: notifica y gestiona la ubicación de los nuevos PODs cuando un nodo queda fuera de servicio.
- **Replicación**: mantiene el número correcto de PODs, evitando que haya más de los indicados en la configuración.

Por último, el etcd es un tipo de almacenamiento distribuido basado en clave-valor. Es el encargado de almacenar el estado del cluster y sus correspondientes PODs.

2.2.5 – Terraform

Terraform es una herramienta de código abierto que permite definir nuestra infraestructura como código. Esto nos permite automatizar el proceso de despliegue de la arquitectura como si de un programa se tratara.

Principales beneficios del uso de Terraform:

- **Portabilidad**: la gran ventaja de Terraform es la facilidad de replicar infraestructuras para un mismo proveedor (Amazon, Google…). Terraform está basado en módulos configurables que permiten modificar de forma sencilla las especificaciones de cada engranaje de la arquitectura.
- **Facilidad a la hora de administrar grandes infraestructuras.**
- **Graficable:** Antes de levantar ninguna arquitectura, Terraform dibuja un plan a seguir. De esta forma, se facilita el control de errores y dependencias cíclicas.

La unidad principal de Terraform es el módulo. Cada módulo, semejante a una clase en POO, contiene recursos relacionados entre sí. De esta forma permitimos que cada módulo cumpla una función determinada.

2.3 – Entregables

Debido al carácter privado del código realizado durante la estancia en la empresa SDG Consulting España, únicamente será entregada la memoria del proyecto.

A lo largo de la memoria, aparecen diferentes piezas código con la intención meramente informativo y aclarativa.

2.4 – Definición técnica del estado del arte y de la solución

A continuación, se detallan los aspectos para tener en cuenta durante la realización del proyecto. Se muestra el papel que tienen los kernel.json, programas que permiten ejecutar diferentes lenguajes a través de protocolos propios.

2.4.1 - Papel que juegan los archivos de especificación de kernel (kernel.json) en el Jupyter Notebook por defecto

Un kernel se identifica mediante la creación de un directorio. El nombre del directorio se usa como identificador del kernel. Dentro de dicho directorio se encuentra el fichero kernel.json. Es el encargado de decir a Jupyter como debe lanzar el kernel. Este fichero contiene una estructura marcada en forma de diccionario:

- Argv → comando con el que vamos a lanzar el kernel.
- Display_name → Nombre que se va a mostrar al usuario.
- Language → Python
- Env → Variables de entorno que se pasan al kernel de Python.
- Metadatos → Información extra añadida al kernel.
- Interrupt_mode (por defecto signal) → como se interrumpe la ejecución de una celda por parte del cliente

Por defecto estos kernel.json pueden encontrarse en diferentes localizaciones. Para el caso de sistemas UNIX, pueden encontrarse en los directorios indicados en la tabla 2:

System	/usr/share/jupyter/kernels
	/usr/local/share/jupyter/kernels
Env	{sys.prefix}/share/jupyter/kernels
User	~/.local/share/jupyter/kernels

Tabla 11 - Directorios de los kernel en un entorno Unix

2.4.2 – Acciones efectuadas por la extensión nb_conda_kernels para modificar la búsqueda de kernels

El paquete nb_conda_kernels modifica el comportamiento por defecto de la clase KernelSpecManager. Esta clase modifica dinámicamente las especificaciones del kernel (kernelspec) para que pueda ejecutarse en un entorno específico de forma correcta.

Cuando creamos un entorno conda e instalamos el ipykernel, en la carpeta /anaconda3/envs/{nombre_env}/share/jupyter/kernels/{directorio} podemos encontrar el kernel.json que más tarde encuentra el notebook_server. La ilustración 8 muestra un kernel python ejecutado en local usando el binario del entorno conda.

```
{
 "argv": [
  "/home/luis/anaconda3/envs/test_env_1/bin/python",
  "-m",
  "ipykernel_launcher",
  "-f",
  "{connection_file}"
 ],
 "display_name": "Python 3 - Test Env 1",
 "language": "python"
}
```

Ilustración 8 - Estructura de un kernel.json por defecto

El fichero de conexión lo podemos encontrar en el directorio que muestra el comando jupyter –runtime-dir. El fichero contiene información sobre los puertos, protocolo y claves (ver Ilustración 8) necesaria para establecer la conexión.

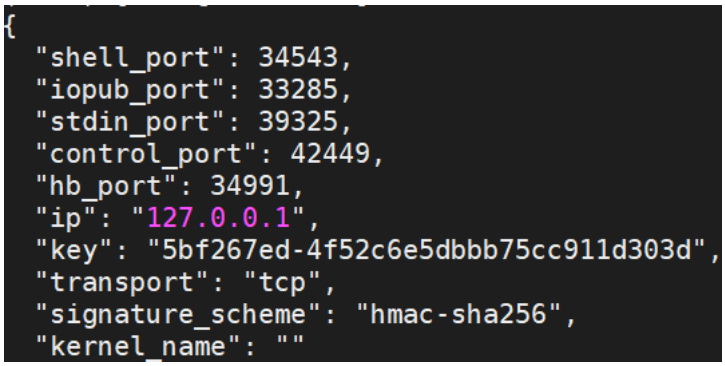


Ilustración 9 - Estructura de un fichero de conexión JN

Cuando lanzamos el notebook *jupyter)* se carga un fichero de configuración */anaconda3/etc/jupyter/jupyter_notebook_config.json* que pone como kernel_spec_manager a **CondaKernelSpecManager.**

Dicha clase obtiene los directorios donde están los entornos conda y los guarda en un diccionario donde la clave es el nombre y el valor la ruta.

conda_info[env] = ["/home/luis/anaconda3", /home/luis/anaconda3/envs/test_env_1", "/home/luis/anaconda3/envs/test_env_2"]

A partir de esta variable obtiene el diccionario del kernel.json

2.4.3 – Papel que juegan los archivos de especificación de kernel en Jupyter Enterprise Gateway

La forma de decir al sistema cómo tiene que manejar un kernel es a través de su especificación (kernel spec). Estos kernel incluyen metadatos necesarios para establecer la conexión con el cluster de Kubernetes.

Al contrario que los kernel.json que crea conda, estos no tienen entre los argumentos el `launch_ipykernel`, sino que llaman a un script que levanta un contenedor, y este último, al acabar de iniciarse sí que ejecuta el kernel de IPython.

2.4.4 – Comunicación entre Jupyter Notebook y Jupyter Enterprise Gateway actual

En el momento en el que el usuario selecciona un kernel, se realiza una petición POST a Jupyter Enterprise Gateway con la información necesaria para levantar el kernel.

Una vez recibida la información necesaria (JSON con las especificaciones del kernel), Jupyter Enterprise Gateway ejecuta el script encargado de levantar el POD en el cluster de Kubernetes. Dicho script, `launch_kubernetes.py`, se encarga traducir el JSON recibido en un objeto POD, estableciendo la configuración deseada.

En el momento que el POD se levanta y queda listo para recibir peticiones, se establece una comunicación entre Jupyter Notebook y el mismo.

2.4.5 – Solución final

El objetivo de este proyecto es realizar un KernelSpecManager cuya función se asemeje a la clase CondaKernelSpecManager. De esta forma, cada usuario tendrá en un directorio NFS su directorio de usuario, y dentro este, los entornos conda propios.

Cada uno de estos entornos conda se mostrarán en la interfaz de Jupyter Notebook, incluidos una serie de entornos predefinidos por el administrador de la plataforma.

Para crear los entornos conda, se realizará un script de creación que adecue la estructura del kernel.json para poder ser usado en un entorno remoto de Kubernetes.

Para realizar el filtrado por nombre de usuario se modificarán las librerías de Jupyter Notebook y Jupyter Enterprise Gateway con el fin de permitir dicha acción. Además, al ser una funcionalidad que da un valor añadido al paquete Jupyter Enterprise Gateway se realizará una Pull Request en su página oficial de GitHub.

La siguiente arquitectura será la imagen sobre la que basar nuestros avances hasta conseguirla:

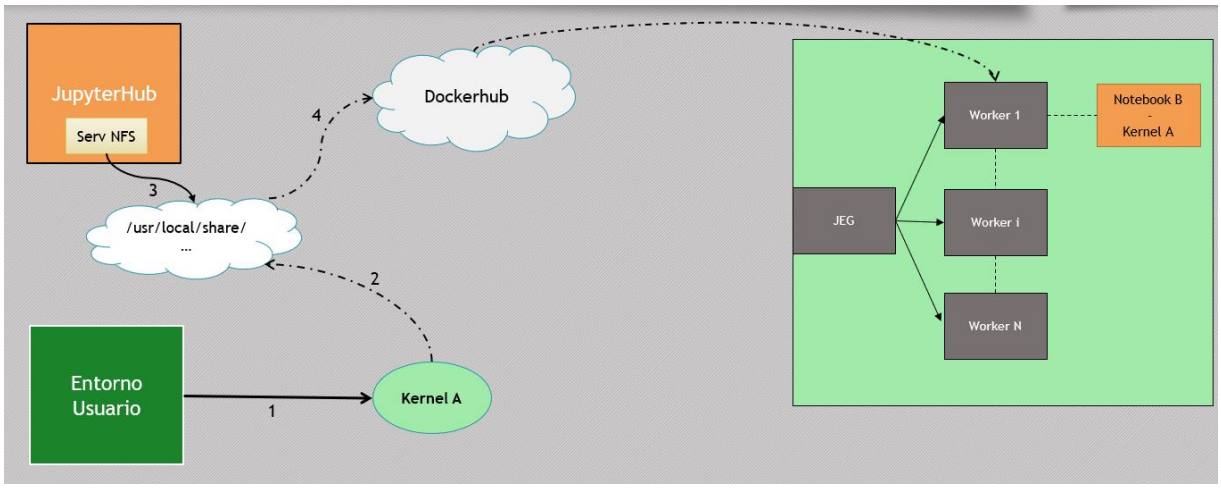


Ilustración 10 - Ejemplo de arquitectura final

3 – Diseño

3.1 – Planteamiento inicial

El objetivo es tener una arquitectura portable (vía Terraform) que permita lanzar cargas de notebooks contra un cluster de Kubernetes. Todos los ficheros se almacenarán en un sistema EFS que permita su recuperación más tarde.

Para lograr los avances, y basándonos en la metodología elegida, vamos a dividir el proyecto en 3 incrementos. Cada uno de estos incrementos tratará de añadir valor al proyecto cumpliendo los objetivos pertinentes.

3.2 – Arquitectura en Amazon Web Services

El dibujo de la arquitectura de la plataforma es el siguiente:

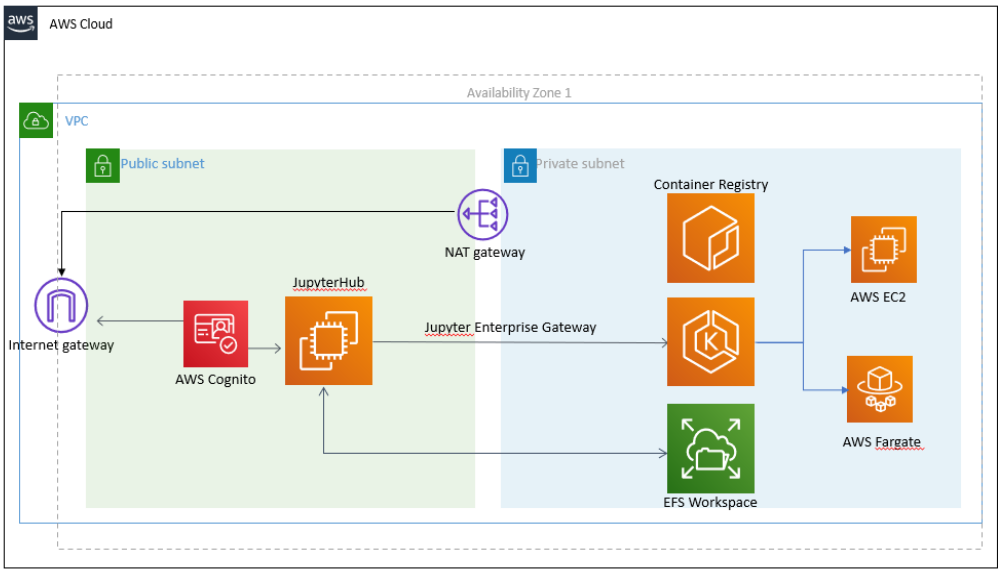


Ilustración 11 - Arquitectura en AWS

Esta arquitectura estará alojada dentro de una red con rango 10.98.0.0/16. Esta red contendrá dos redes privadas (10.98.2.0/24 y 10.98.3.0/24) y una red pública (10.98.1.0/24).

La subred pública, con salida a internet gracias a la Internet Gateway, contiene alojada la máquina EC2 de JupyterHub. Esta máquina tendrá configurado el AWS Cognito como sistema de autenticación. De esta forma, se permite al administrador de la plataforma gestionar la creación/eliminación de usuarios.

Dentro de las subredes privadas estarán alojados los siguientes elementos propios de Amazon:

- **EKS**: Es el servicio de Kubernetes gestionado por Amazon. Permite al desarrollador evitar el proceso inicial de configuración de cluster, dando al usuario un cluster ya configurado y listo para su uso.
- **Container Registry**: es el repositorio de imágenes propias de Amazon. De esta forma podremos subir nuestras imágenes Docker a dicho repositorio en vez de a uno público como es DockerHub.
- **EFS Workspace:** Permite compartir directorios/ficheros entre diferentes máquinas. Trabaja al igual que un servidor NFS, permitiendo montar la unidad EFS en cualquier máquina.

- **AWS EC2:** Maquinas proporcionadas por Amazon. Serán las encargadas de alojar los diferentes PODs del sistema. Harán el rol de nodo esclavo.
- **NAT Gateway:** permite la salida a internet de las redes privadas.

3.3 – Jupyter Enterprise Gateway

El paquete de utilidades Jupyter Enterprise Gateway no permite la gestión de entornos por usuario. Esto limita las posibilidades de su uso en proyectos empresariales, ya que únicamente permite aquellos kernel que se encuentran predefinidos en una de las carpetas específicas para kernels de Jupyter.

Ante esta situación, y con el conocimiento de la existencia de una extensión de Jupyter como es nb_conda_kernels, se ha decidido modificar el comportamiento de JEG para habilitar la gestión de usuarios.

JEG permite añadir nuestros propios kernels a través de un servicio NFS que monta el POD encargado de buscarlos. Por ello, el NFS estará definido por la siguiente estructura:

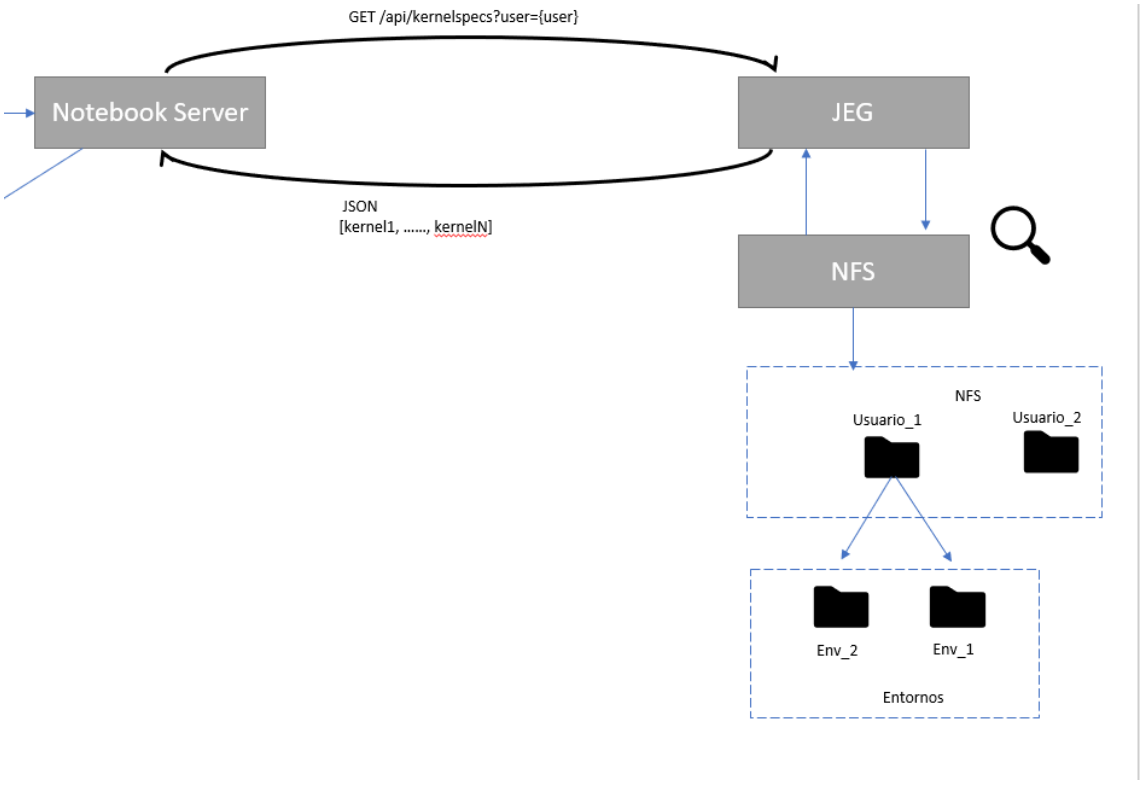


Ilustración 12 - Flujo de trabajo Jupyter Enterprise Gateway

La clase encargada de buscar los kernels disponibles es *KernelSpecManager.* Dicha clase no trabaja con parámetros de ningún tipo, por lo que habrá que crear una clase *UserKernelSpecManager* que sí permita dichos parámetros y cuyos métodos sean llamados desde las clases *MainKernelSpecHandler* y *KernelSpecHandler*

MainKernelSpecHandler es la clase encargada de pedir a Jupyter Enterprise Gateway los kernels disponibles. Como respuesta, recibe un diccionario en formato JSON con todas las especificaciones de cada kernel.

KernelSpecHandler se encarga de pedir las especificaciones de un único kernel. Esta llamada se produce una vez elegido el kernel a desplegar.

El usuario, una vez logueado en JupyterHub, realizará una petición del tipo /api/kernelspecs?user=<user_name> y JEG le devolverá una lista de los entornos conda creados por dicho usuario.

Un error al que se está expuesto es la probabilidad de fallo en determinadas librerías debido a que son compiladas por entes diferentes (máquina donde ha sido instalada y máquina donde se ejecuta) como puede ser TensorFlow. Para evitar esto, se ha decidido brindar al usuario una serie de kernels ya predefinidos con dichas librerías.

Cuando se reciba una petición de creación de kernel, Jupyter Enterprise Gateway consultará a una base de datos local (a la máquina de JupyterHub) si dicho usuario y entorno tienen permisos para ello. Si el resultado de la comprobación es favorable, la consulta devolverá una tupla con los máximos y mínimos de CPU y RAM permitidos. En ese momento, Jupyter Enterprise Gateway levanta el contenedor haciendo uso del script de Python `launch_kubernetes.py`

Para establecer dichos límites se ha diseñado una pequeña base de datos cuyo diseño E/R está detallado en la Ilustración 13.

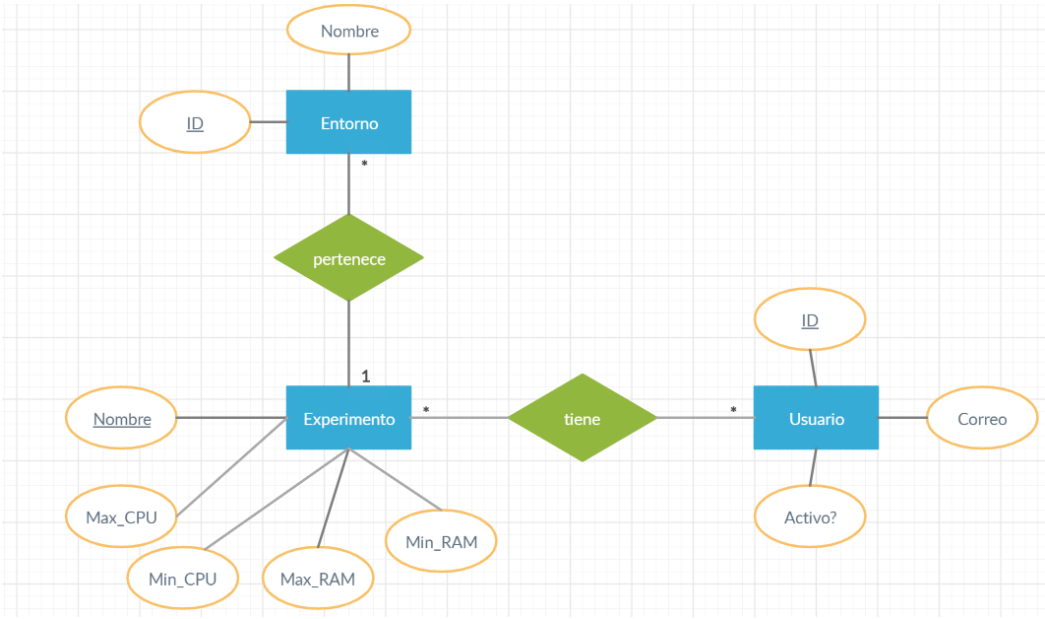


Ilustración 13 – Esquema E/R de la BD

Un usuario, identificado por un ID, puede estar en varios experimentos, o proyectos, al mismo tiempo. Cada experimento va a tener asociado una serie de entornos, no pudiendo existir un entorno en varios experimentos. De cada usuario guardaremos su correo y un booleano que nos indica si está dado de alta en la plataforma o no.

Cada experimento, identificado por su nombre, va a tener una serie de atributos como son: máximos y mínimos de CPU y RAM. Estos límites corresponden a los valores máximos y mínimos que tendrá cada POD asociado a dicho experimento.

Un entorno, además de estar identificado por un ID, tendrá asociado un nombre.

3.4 – Jupyter Notebook

Jupyter Notebook no permite realizar peticiones añadiendo parámetros de petición. En vista a los cambios que se van a realizar en Jupyter Enterprise Gateway (requisito funcional N.º3), se ha de permitir dicho envío de parámetros en la petición.

Para permitirlo, necesitaremos modificar el comportamiento de la clase *GatewayKernelSpecManager* para que añada un parámetro de petición *user* en las comunicaciones con Jupyter Enterprise Gateway. La ilustración 14 representa el comportamiento esperado de Jupyter Notebook.

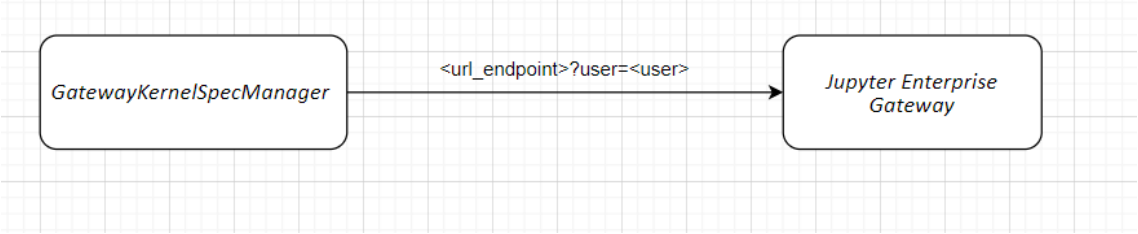


Ilustración 14 - Comportamiento esperado entre clases

Dicha nueva clase, tendrá un método estático que será llamado en el constructor para asignar al atributo `base_endpoint` el parámetro de petición *user.*

De esta forma, ya tendríamos configurado el canal de comunicación para el traspaso de parámetros entre el cliente (Jupyter Notebook) y el servicio REST (Jupyter Enterprise Gateway).

3.5 – Pull Request a Jupyter

Para asegurarse que el funcionamiento actual, tanto de Jupyter Notebook como de Jupyter Enterprise Gateway, va a seguir siendo el mismo, hay que asegurarse de tratar esta nueva funcionalidad como algo opcional.

De esta forma, si el desarrollador está trabajando con una versión antigua de Jupyter Notebook, que no soporta este tipo de peticiones filtradas, va a seguir recibiendo todos los kernels disponibles.

De la misma forma, si se tiene una versión de Notebook que permite dicha funcionalidad, pero no se trabaja con la versión adecuada de Jupyter Enterprise Gateway, este último ignorará los parámetros de petición enviados por Jupyter Notebook.

4 – Implementación

4.1 - Creación de un cluster de Kubernetes con 2 nodos esclavos usando máquinas virtuales

4.1– Prerrequisitos

1. VirtualBox: software que te permite crear máquinas virtuales hospedadas sobre una máquina física con un sistema operativo anfitrión.
2. CentOS 7 ISO: imagen ISO correspondiente a la distribución CentOS basada en Linux. He elegido la versión *minimal* ya que contiene los paquetes mínimos sin interfaz gráfica.

4.2– Configuración e instalación

El proceso de configuración e instalación inicial de los paquetes necesarios para la puesta a punto del cluster de Kubernetes es común para todo tipo de nodo (maestro y esclavo).

En primer lugar, y una vez operativa la máquina virtual, debemos instalar Docker y Kubernetes en nuestros nodos. Para ello debemos deshabilitar antes dos propiedades de seguridad: SELinux y el cortafuegos (*firewall*).

SELinux define controles de seguridad para aplicaciones, procesos y ficheros que residen en nuestro sistema. Usa políticas de seguridad, conjunto de reglas que indican a SELinux que puede ser accesible.

Cuando una aplicación o proceso, conocido como agente, realiza una petición a un objeto, como un fichero, SELinux comprueba si dicho agente tiene los permisos necesarios para acceder a dicho objeto.

Para deshabilitar dicha comprobación previa, debemos editar el fichero de configuración /etc/selinux/config:

```
SELINUX=permissive
```

El cortafuegos, o *firewall* en inglés, es el encargado de bloquear comunicaciones a través de puertos a agentes no autorizados, permitiendo al mismo tiempo comunicaciones autorizadas. Deshabilitándolo estaremos permitiendo todo tipo de comunicación vía puerto.

Para deshabilitarlo basta con introducir el siguiente comando:

```
systemctl disable firewalld && systemctl stop firewalld
```

Una vez hayamos instalado los paquetes necesarios (docker, kubelet, kubeadm, kubectl, Kubernetes-cni), debemos habilitar docker y kubelet como servicios propios del sistema:

```
systemctl enable docker && systemctl start docker
systemctl enable kubelet && systemctl start kubelet
```

Por último, deberíamos deshabilitar Swap. Swap es un espacio de intercambio, que bien puede ser una partición lógica o un archivo. La función principal de swap es llevar los datos que se encuentran en memoria RAM a un archivo, de forma que liberamos espacio en la memoria RAM. De esta forma, el sistema tendrá una mayor memoria que únicamente la física. A esta combinación entre memoria RAM y swap se conoce como memoria virtual.

Una vez realizados estos cambios ya tendríamos lista la máquina virtual. Para evitar repetir el proceso de configuración se recomienda clonar la máquina virtual.

Para cada nodo esclavo, tenemos que modificar su hostname y modificar en fichero /etc/hosts añadiendo por cada nodo del clúster una entrada con el siguiente formato:

```
hostname_nodo <ip-nodo>
```

4.3 – Configuración del panel de control de Kubernetes en el nodo maestro

Para la configuración del nodo maestro y unión al cluster de los nodos esclavos, vamos a utilizar la herramienta kubeadm.

Kubeadm realiza las acciones necesarias para levantar y mantener operativo un clúster de Kubernetes. Por definición, únicamente se encarga del lanzamiento del proceso y no del aprovisionamiento de las máquinas (sección anterior).

El comando encargado de inicializar el panel de control es *kubeadm init*. Al finalizar dicho proceso, nos mostrará por pantalla el comando necesario para adherir nuevos nodos esclavos a nuestro cluster.

```
kubeadm join <ip-master-node>:6443
--token <token>
--discovery-token-ca-cert-hash <sha256-hash>
```

Una vez introducido este comando en los distintos nodos esclavos ya tendríamos estructurado el cluster de Kubernetes. Para que todos los nodos puedan comunicarse entre ellos, necesitamos instalar una librería que lo permita. Estas librerías son conocidas como Container Network Interface (CNI). El CNI proporciona una API para conectar los contenedores, ejecutados dentro del cluster, con la red externa.

Existen numerosos tipos de CNI en función de la plataforma donde quieras desplegar el cluster.

En la siguiente tabla podemos ver una comparación entre diferentes CNIs en función de su dificultad de instalación, nivel de seguridad, actuación y optimización de recursos.

CNI	INSTALL	SECURITY	PERFS	RESOURCES
Calico				
Canal				
Cilium				
Flannel				

Tabla 12 - Comparación entre CNIs

En nuestro caso hemos elegido Calico. Proporciona un alto nivel de seguridad, de optimización de recursos y es sencillo de instalar.

Para su instalación basta con ejecutar el siguiente comando en el nodo maestro de nuestro cluster. De esta forma, permitimos la comunicación entre los distintos elementos que forman el cluster de Kubernetes, tanto lógicos (POD, Container) como físicos (nodos esclavos, nodo maestro)

```
kubectl apply -f https://docs.projectcalico.org/v3.5/getting-started/kubernetes/installation/hosted/etcd.yaml

kubectl apply -f https://docs.projectcalico.org/v3.5/getting-started/kubernetes/installation/hosted/calico.yaml
```

4.3– Instalación del dashboard de Kubernetes

Kubernetes proporciona una forma visual de controlar todos los recursos y despliegues que forman el cluster de Kubernetes. Permite trabajar con una interfaz gráfica, monitorizar los recursos empleados de cada nodo esclavo e interactuar con los procesos en ejecución.

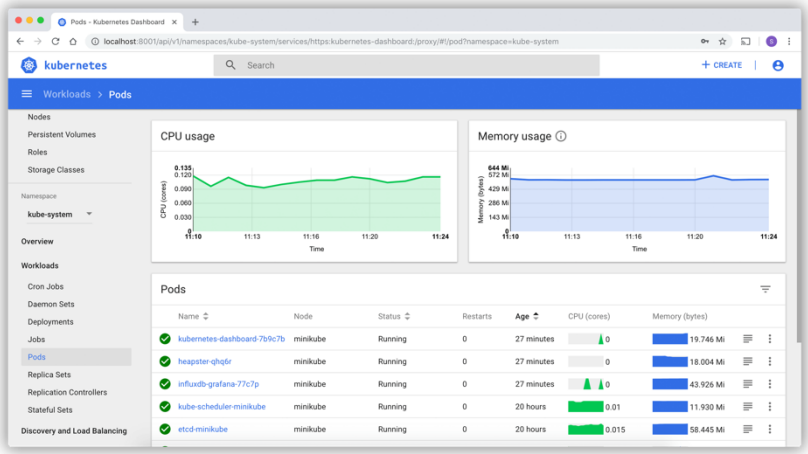


Ilustración 15 - Vista del panel de control de Kubernetes

Para la instalación de dicha interfaz, tenemos que lanzarla como servicio propio del cluster. Para ello:

```
kubectl create -f https://raw.githubusercontent.com/kubernetes/dashboard/master/aio/deploy/recommended/kubernetes-dashboard.yaml
```

Una vez desplegada la interfaz web, necesitamos un token para poder iniciar sesión como administradores. Autenticándonos como administradores, tendremos todos los permisos disponibles.

Para crear un token administrador tenemos que crear un rol administrador dentro del cluster de Kubernetes.

```
kubectl --namespace kube-system create serviceaccount k8s-admin
kubectl create clusterrolebinding k8s-admin --serviceaccount=kube-system:k8s-admin --clusterrole=cluster-admin
```

4.2 – Modificación del paquete Jupyter Enterprise Gateway

Una vez configurado y listo el entorno de trabajo, hay que modificar el paquete de Jupyter Enterprise Gateway con el fin de ajustarlo a los requisitos establecidos.

4.2.1 - Comportamiento actual

Por defecto, Jupyter Enterprise Gateway devuelve todos los kernels que encuentra en el directorio NFS `/usr/local/share/jupyter/kernels`. De esta forma, si queremos añadir nuevos kernels bastaría con añadirlos en el primer nivel de este directorio.

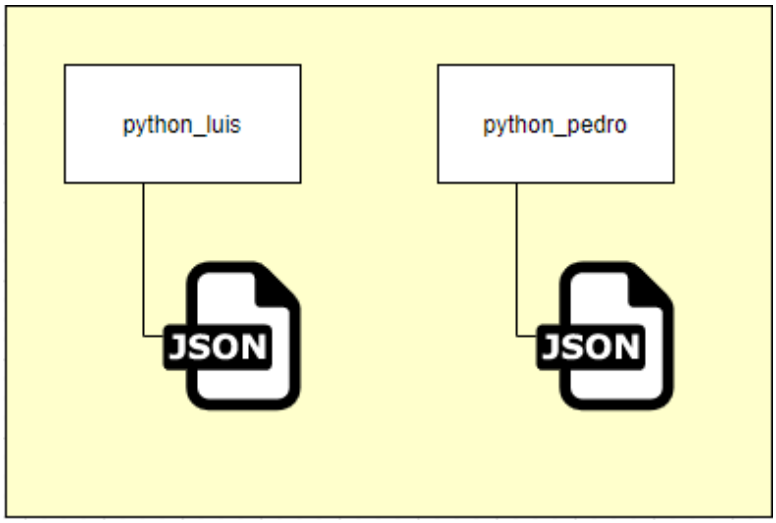


Ilustración 16 - Estructura NFS predeterminado

Aunque existen alguna forma de filtrar los kernels, estos deben estar predefinidos, privando al usuario del control de estos.

4.2.2 - Comportamiento deseado

El directorio NFS tendrá en un primer nivel el directorio *home* asociado a cada usuario de la plataforma. Cada uno de estos directorios contendrá, entre otros, un directorio que contenga los entornos conda creados por él mismo. Cada uno de esos entornos, en el subdirectorio `<entorno>/share/jupyter/kernels/python3`, contiene el fichero kernel.json.

Cuando se reciba una petición con el parámetro de petición user=<usuario>, Jupyter Enterprise Gateway buscará todos los ficheros kernel.json dentro del directorio home del usuario.

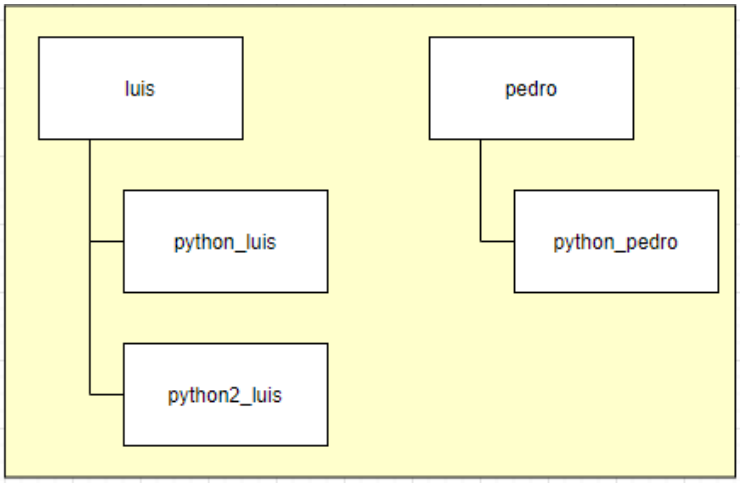


Ilustración 17 - Estructura NFS por usuario

4.2.3 - Primer incremento

El problema expuesto radica en la necesidad de un parámetro que permita a Jupyter Enterprise Gateway devolver únicamente los kernels pertenecientes al usuario.

Basándome en *handlers* (manejadores) similares para diferentes elementos, como pueden ser los propios para lanzar/parar/apagar kernels, decidí crear una nueva clase *KernelSpecHandler* cuyos métodos permitieran un parámetro de entrada denominado *kernel_user.* Este parámetro determinaría qué kernels podría tener un cierto usuario.

Esta nueva clase modificaría los métodos heredados de la clase padre *(APIHandler)* con los cambios comentados anteriormente. La nueva clase, llamada UserKernelSpecHandler, modificaría el método *get* con el fin de permitir la entrada de un parámetro.

```
class UserKernelSpecHandler (APIHandler):
        @web.authenticated
        @gen.coroutine
        def get(self, kernel_user = None):
            ksm = self.kernel_spec_manager
            km = self.kernel_manager
            model = {}
            model['default'] = km.default_kernel_name
            model['kernelspecs'] = specs = {}
            kspecs = yield maybe_future (ksm.get_all_specs ())
            if kernel_user:
                self.log.info("Searching kernels for user '%s' " %kernel_user)
            else:
                self.log.info("No user. All kernels given")
            for kernel_name, kernel_info in kspecs.items ():
                try:
                    if is_kernelspec_model (kernel_info):
                        d = kernel_info
                    else:
                        d = kernelspec_model(self, kernel_name, kernel_info['spec'],
    kernel_info['resource_dir'])
                    d = apply_user_filter(d, kernel_user)
                    if d is not None:
                        self.log.info("Find kernel '%s'", d['name'])
                        specs[kernel_name] = d
                except Exception:
                    self.log.error("Failed to load kernel spec: '%s'", kernel_name)
                    continue
            self.set_header("Content-Type", 'application/json')
            self.finish(json.dumps(model))
```

Dicho método recibe como parámetro de función un nombre de usuario. Este parámetro se ha puesto como opcional con el fin de mantener operativo el comportamiento entre Jupyter Notebook y Jupyter Enterprise Gateway, en caso de versiones más antiguas de uno u otro.

Dentro del método se realiza una llamada a la función *apply_user_filter*. Esta función se ha creado *exprofeso* para la filtración de los kernels dado un nombre de usuario.

```
def apply_user_filter(kernelspec_model, kernel_user = None):
      if kernel_user:
        # Check unauthorized list
        if exists(kernelspec_model, ['spec', 'metadata', 'process_proxy', 'config',
    'unauthorized_users']):
          # Check if kernel_user in kernelspec_model
          unauthorized_list =
    kernelspec_model['spec']['metadata']['process_proxy']['config']['unauthorized_use

          if kernel_user in unauthorized_list:
            return None
        # Now we have check that unauthorized list doesnt exist
        if exists(kernelspec_model, ['spec', 'metadata', 'process_proxy', 'config',
    'authorized_users']):
          authorized_list =
    kernelspec_model['spec']['metadata']['process_proxy']['config']['authorized_users

          if authorized_list and kernel_user not in authorized_list:
            return None
      return kernelspec_model
```

Este método comprueba que dado un kernelspec_model, existe el elemento kernel_user dentro de la lista de usuarios autorizados.

Un kernelspec_model es un diccionario con campos anidados que contiene información sobre un kernel. Este objeto hace referencia a los ficheros kernel.json que hemos hablado anteriormente.

La función devolverá un resultado diferente a None cuando:

- No se haya especificado ningún usuario
- No exista ninguna lista de usuarios autorizados ni no autorizados
- Se encuentre únicamente en la lista de usuarios autorizados

Durante la ejecución de este método cabe destacar la llamada a la función *exists*, que permite buscar parejas clave-valor en diccionarios anidados.

```
def exists(obj, chain):
      _key = chain.pop(0)
      if _key in obj:
          return exists(obj[_key], chain) if chain else obj[_key]
```

Una vez creado el nuevo manejador, debemos añadirlo a la aplicación para poder realizar peticiones contra esa URL.

Antes hay que definir la nueva ruta que vamos a añadir a nuestra aplicación. Para ello, he asignado a diferentes URLs, el mismo manejador. De esta forma, evitamos modificar la funcionalidad previa y añadimos una extra.

```
kernel_user_regex = r"(?P<kernel_user>[\w%]+)"
kernel_name_regex = r"(?P<kernel_name>[\w\.\-%]+)"
default_handlers_notebooks = [
            (r"/api/kernelspecs", ModifyKernelSpecHandler),
            (r"/api/kernelspecs/user/%s" % kernel_user_regex, ModifyKernelSpecHandler
            (r"/api/kernelspecs/%s" % kernel_name_regex, KernelSpecHandler)
]
```

Para comprobar su correcto funcionamiento, vamos a probarlo con la siguiente estructura de ficheros (ruta raíz -> /usr/local/share/Jupyter/kernels):

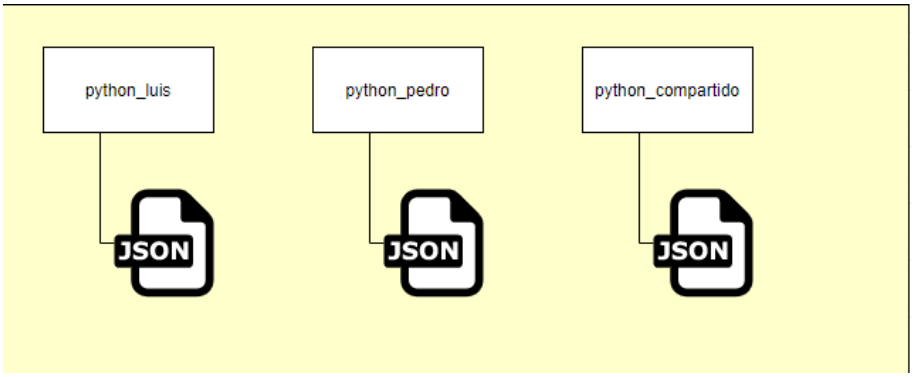


Ilustración 18 - Estructura NFS predeterminado

Cada JSON contiene un campo anidado *authorized_users* que limita el uso del kernel a ciertos usuarios. En cambio, el kernel de *python_compartido* carece de dicho campo, siendo público para todos los usuarios.

Desde la terminal, y con la aplicación lanzada, realizamos una petición HTTP contra la nueva ruta de Jupyter Enterprise Gateway:

```
curl http://127.0.0.1:8888/api/kernelspecs/user/luis
```

Obteniendo como resultado un objecto JSON:


```
{
    "default": "python3",
    "kernelspecs": {
        "python3": {
            "name": "python3",
            "spec": {
                "argv": [
                    "python",
                    "-m",
                    "ipykernel_launcher",
                    "-f",
                    "{connection_file}"
                ],
                "env": { },
                "display_name": "Python 3 desde PATH-test",
                "language": "python",
                "interrupt_mode": "signal",
                "metadata": {
                    "process_proxy": {
                        "config": {
                            "authorized_users": [
                                "luis"
                            ]
                        }
                    }
                }
            },
            "resources": {
                "logo-64x64": "/kernelspecs/python3/logo-64x64.png",
                "logo-32x32": "/kernelspecs/python3/logo-32x32.png"
            }
        }
    }
}
```


Los principales problemas de esta solución son:

- Se ha añadido una nueva ruta a la API (/api/kernelspecs/user/<user>) complicando la interacción entre rutas. Esto provoca un mal mantenimiento del código.
- Necesidad de que los kernels tengan los campos adecuados. Esto impide liberar al usuario de sus kernels ya que tiene que modificarlos (o un administrador)
- Lentitud a la hora de buscar kernels. Si hay 10.000 kernels, va a tener que comprobar que todos si el usuario está autorizado o no.

4.2.4 - Segundo incremento

Al acabar la primera iteración tenemos un nuevo sistema de filtrado de kernels. El principal problema que tiene es que dichos kernels deben tener de antemano una serie de campos escritos por el usuario (lista de usuarios autorizados). Además, si el número de kernels es muy alto puede verse mermada la velocidad de ejecución de dicho método.

Para evitar esta situación, y tratar de liberar al usuario/administrador de la gestión de los kernel.json, se ha decidido llevar los directorios HOME de los usuarios a la ruta */usr/local/share/Jupyter/kernels.*

Con este escenario damos una estructura lógica a los kernels, perteneciendo cada directorio kernel al home del usuario (estructura descrita en la imagen 15).

Analizando el código podemos observar que dentro del manejador GET de la clase UserKernelSpecHandler se realiza una llamada a *ksm.get_all_specs()*. Este método se encarga de pedir información sobre los kernels que están disponibles (sin importar el usuario que ha realizado la petición).

Para añadir la funcionalidad de filtrar kernels por usuario, en dicha llamada necesitamos pasarle información de quién la está realizando (parámetro *kernel_user*). Esta nueva clase, hija de la clase KernelSpecHandler, modificará los métodos de la clase padre para añadir un nuevo parámetro de método opcional que será el *kernel_user*. Las nuevas cabeceras de los métodos de dicha clase serán semejantes a las siguientes:

```
def find_kernel_specs(self, kernel_user=None):
```

El método principal de la clase es *get_all_specs*, que es llamado desde el manejador *ModifyKernelSpecHandler,* tiene un parámetro de función optativo llamado *kernel_user.* Este método se encarga de buscar los todos los kernels disponibles para dicho usuario, y devuelve un diccionario de la siguiente forma:

```
{
  'kernel_name': {
    'resource_dir': '/path/to/kernel_name',
    'spec': {"the spec itself": ...}
  }
}
```

Este método, realiza una serie de llamadas anidadas entre distintos métodos que se encargar de encontrar los kernels. El principal método dentro de esta cadena de llamadas es *find_kernel_specs*, que se encarga de buscar los kernels en una serie de directorios predeterminados.

Cuando se encuentre en el directorio */usr/local/share/Jupyter/kernels* en vez de buscar en el primer nivel (como hace de forma predeterminada), baja un nivel más en el árbol de directorios haciendo uso del parámetro de función *kernel_user.*

Al acabar dicho incremento podemos notar una mejoría en la búsqueda de kernels ya que únicamente tiene que buscar en el directorio del usuario, sin comprobar que dicho usuario es propietario del kernel (se presupone que es propietario). De esta forma reducimos los directorios donde puede haber kernels a uno sólo: el directorio Home del usuario.

4.2.5 – Tercer incremento

Durante las dos últimas iteraciones hemos ido aumentando las características de Jupyter Enterprise Gateway. Estas características son las siguientes:

- Permitir la recepción de un parámetro de petición llamado *user.*
- Aislar los kernels de forma independiente en función del usuario.

El problema actual radica en que los ficheros JSON hacen referencia a unas imágenes Docker. Estas imágenes ya están creadas, siendo estáticas e impidiendo al usuario instalar nuevas librerías de forma permanente. Esto provoca que un usuario tenga que instalar las librerías necesarias cada vez que inicia un nuevo notebook.

Para paliar este problema tenemos que trabajar con entornos conda. Estos entornos permiten aislar las diferentes versiones de programas, evitando así la incompatibilidad entre programas de diferentes entornos.

Estudiando el comportamiento de Jupyter Enterprise Gateway en el momento en el que se levanta un contenedor con el kernel de Python, podemos ver que al final de la imagen Docker hay una instrucción que se encarga de lanzarlo:

```
USER jovyan
ENV KERNEL_LANGUAGE python
CMD /usr/local/bin/bootstrap-kernel.sh
```

Este script se encarga de lanzar el kernel de Python y establecer la conexión entre el POD (donde se está ejecutando el script) con Jupyter Enterprise Gateway. Por defecto, el kernel de Python se está lanzando, haciendo uso del comando Python.

```
launch_python_kernel() {
    export JPY_PARENT_PID=$$
    set -x
    python ${KERNEL_LAUNCHERS_DIR}/python/scripts/launch_ipykernel.py . . .
    { set +x; } 2>/dev/null
    }
```

Este comando viene preinstalado en el contenedor, por lo que solo se van a utilizar las librerías que se encuentren en la carpeta *bin* de ese Python. Si queremos que se ejecute el binario Python correspondiente a nuestro entorno conda, debemos ejecutar el script haciendo uso del binario Python de nuestro entorno conda.

```
launch_python_kernel() {
    export JPY_PARENT_PID=$$
    set -x
    /path/to/user/conda/bin/python ${KERNEL_LAUNCHERS_DIR}/python/scripts/launch_ipykernel.py
    { set +x; } 2>/dev/null
    }
```

De esta forma, cualquier librería que instalemos en nuestro entorno conda, vamos a poder usarla en el notebook sin tener que instalar nada en el contendor correspondiente. Para generalizar la ruta, en vez de usar la absoluta trabajaríamos con variables de entorno.

Por último, faltaría limitar los recursos del nodo en función del proyecto sobre el que se va a trabajar (Cód. Req Fun. 4). Esta consulta, como hemos visto en el análisis y diseño, se realiza en el momento en el que se crea el POD vía el script `launch_kubernetes.py`.

Este script se conectará a la base de datos SQL almacenada en la máquina de JupyterHub. Para permitir a Jupyter Enterprise Gateway conectarse, necesitamos saber de antemano la IP de JupyterHub. Esta variable será pasada vía variables de entorno en el fichero YAML correspondiente a JEG.

La consulta tendrá que comprobar que, dado un usuario y un entorno seleccionado, existe una asociación entre ambos. Dicha asociación, única, deberá tener los atributos relacionados con el experimento asociado a ese entorno. En caso de que la consulta no devuelva ninguna tupla, se devolverá None con el fin de provocar un fallo. Esto impide usar entornos a usuarios no autorizados.

```
# Cogemos el valor de KERNEL_ENV y KERNEL_USER
kernel_env = os.environ.get('KERNEL_ENV_EXP')
kernel_user = os.environ.get('KERNEL_USER')

# Buscamos si un usuario determinado pertenece a un experimento determinado
consulta_sql = "SELECT EXPERIMENTO.* FROM ENTORNO " \
               "INNER JOIN EXPERIMENTO ON ENTORNO.EXPERIMENTO = EXPERIMENTO.ID " \
               "INNER JOIN EXPERIMENTO_USUARIO ON EXPERIMENTO.ID =
EXPERIMENTO_USUARIO.EXPERIMENTO "  \
               "INNER JOIN USUARIO ON EXPERIMENTO_USUARIO.USUARIO = USUARIO.ID " \
               "WHERE ENTORNO.NAME='{}' AND USUARIO.USUARIO='{}'".format(kernel_env,
kernel_user)

cursor.execute(consulta_sql)
usuario_permitido = cursor.fetchone()
```

Una vez tenemos los recursos de dicho POD, tenemos que actualizar el fichero base sobre el que trabaja este script. La plantilla *kernel-pod.yaml.j2* permite dar la misma estructura a todos los PODs.

Durante la ejecución del script `launch_kubernetes.py` se crea un objeto POD cuyos atributos se plasman en el fichero *kernel-pod.yaml.j2,* sustituyendo los valores necesarios.

En nuestro caso, tenemos que añadir los campos relacionados con la gestión de recursos. Esto provocará que, de forma dinámica, los valores entre corchetes {{ }} se sustituyan por los valores pasados como parámetros.

```
resources:
  limits:
    cpu: "{{ max_cpu }}"
    memory: "{{ max_ram }}"
  requests:
    cpu: "{{ min_cpu }}"
    memory: "{{ min_ram }}"
```

4.3 – Modificación del paquete Jupyter Notebook

Una vez modificado Jupyter Enterprise Gateway para que sea capaz de interpretar el parámetro de petición *user*, hay que modificar el cliente (Jupyter Notebook) para que realice las peticiones pertinentes añadiendo dicho parámetro.

Por defecto, Jupyter Notebook trabaja con la variable de entorno *KERNEL_USERNAME*. El valor de esta variable tiene que estar puesto antes de iniciar la aplicación, ya que si no obtendríamos un error de obtención de clave.

El valor de esta variable lo podemos especificar vía línea de comandos haciendo uso del comando *export KERNEL_USERNAME=<value>,* o través de la opción de configuración de Jupyter Notebook --Gateway.http_client.

Para evitar que el usuario tenga que establecer el valor de la variable de entorno, he modificado el comportamiento de Jupyter Notebook para que, en caso de no existir ningún valor asociado a *KERNEL_USERNAME*, use el valor asociado a la variable de entorno *USER.*

```
if os.environ.get('KERNEL_USERNAME') is None and GatewayClient.instance().http_user:
    os.environ['KERNEL_USERNAME'] = GatewayClient.instance().http_user


if os.environ.get('KERNEL_USERNAME') is None and os.environ.get('USER'):
    os.environ['USER'] = GatewayClient.instance().http_user
```

La comunicación que se produce entre Jupyter Notebook y Jupyter Enterprise Gateway para obtener los kernels y lanzarlos, se realiza a través de objetos de la clase *GatewayKernelSpecManager*.

Esta clase tiene como atributo una URL que apunta a la ruta asociada a la gestión de kernels de Jupyter Enterprise Gateway. Todos los métodos de dicha clase realizan una petición contra esa URL.

Como hemos visto antes, hemos añadido la posibilidad de mandar parámetros de petición a la ruta /api/kernelspecs, de forma que cualquier petición se realizaría siguiendo el siguiente patrón:

```
http://<url_enterprise_gateway>:<puerto>/api/kernelspecs?user=<user>
```

Para que todas las peticiones desde esta clase añadan el sufijo *?user=<user>,* creamos un método estático que añada dicho sufijo al atributo de clase *base_endpoint* que llamaremos durante la construcción del objeto.

```
@staticmethod
def _get_endpoint_for_user_filter(default_endpoint):
    kernel_user = os.environ.get('KERNEL_USERNAME')
    if kernel_user:
        return '?user='.join([default_endpoint, kernel_user])
    return default_endpoint
```

4.4 – Funcionamiento en un entorno de Kubernetes

Una vez creado el clúster de Kubernetes, necesitamos crear las nuevas imágenes Docker que contengan la librería Jupyter Enterprise Gateway modificada.

4.4.1 - Creación de imagen Docker con Jupyter Enterprise Gateway

Recordemos que una imagen Docker es un conjunto de directrices que permiten a una "máquina virtual" llegar a un determinado estado. Esto nos permite aislarla del entorno de ejecución, pudiendo ser ejecutada en cualquier maquina con Docker instalado.

El nuevo Dockerfile, fichero a partir del cual se generan las imágenes Docker, consta de una serie de directrices que tienen una función específica:

- **FROM**: esta directriz permite heredar el estado final (usuarios creados, directorios, librerías instaladas) de otra imagen.
- **ENV**: establecer variables de entorno al sistema
- **RUN**: ejecuta los comandos y guarda una copia del estado actual. Cada vez que usamos RUN se genera una imagen Docker temporal cuyo estado es el resultado de ejecutar dichos comandos.
- **USER:** por defecto, todas las ejecuciones provienen del usuario root. Si queremos cambiar de usuario aplicamos dicha directriz.
- **COPY:** copia los archivos/directorios locales dentro de la ruta especificada de la imagen.
- **ADD:** Se diferencia de COPY en la forma de copiar los ficheros. Utilizando ADD sobre un fichero comprimido, se encarga de extraerlos de forma automática.
- **CMD:** Ejecuta un comando/script en tiempo de ejecución de la imagen Docker. La principal diferencia con RUN radica que este último se ejecuta en tiempo de compilación (una única vez cuando se crea la imagen)
- **EXPOSE:** Informa al contenedor que corre dicha imagen sobre qué puerto/protocolo debe escuchar.
- **WORKDIR:** establecer el directorio de trabajo

Las principales diferencias con respecto al Dockerfile que encontramos en GitHub son:

1. En el momento en el que ejecutamos la instrucción `COPY jupyter_enterprise_gateway*.whl` , copiamos el binario del nuevo Enterprise Gateway.
2. El script `start-enterprise-gateway.sh` se ejecuta con permisos root para poder acceder a los diferentes directorios HOME de los usuarios. Si queremos evitar esta falla de seguridad, deberíamos añadir una nueva instrucción que permitiera cambiar de forma recursiva los directorios (esta opción se descartó en su momento debido a la lentitud del proceso)

Dicho script ha sido modificado para que monte una unidad EFS durante la ejecución de este. Para poder pasarle la IP/DNS del servidor NFS se usa la variable de entorno KERNEL_EFS_ID proveniente del fichero kernel.json correspondiente. De esta forma, relacionamos los ficheros kernel.json con las imágenes Docker.

```
env {
      "KERNEL_EFS_ID": "<dns_efs_aws>"
}
```

```
FROM jupyter/minimal-notebook

ENV SPARK_VER 2.4.5
ENV SPARK_HOME /opt/spark
ENV JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
RUN conda install --quiet --yes \
    cffi \
    send2trash \
    requests \
    future \
    pycryptodomex && \
    conda clean -tipsy && \
    fix-permissions $CONDA_DIR && \
    fix-permissions /home/$NB_USER


USER root
RUN apt update && apt install -yq curl openjdk-8-jdk
COPY jupyter_enterprise_gateway*.whl /tmp/
RUN pip install /tmp/jupyter_enterprise_gateway*.whl && \
       rm -f /tmp/jupyter_enterprise_gateway*.whl
ADD jupyter_enterprise_gateway_kernelspecs*.tar.gz /usr/local/share/jupyter/kernels/
ADD jupyter_enterprise_gateway_kernel_image_files*.tar.gz /usr/local/bin/

COPY start-enterprise-gateway.sh /usr/local/bin/
RUN chown jovyan:users /usr/local/bin/start-enterprise-gateway.sh && \
       chmod 0755 /usr/local/bin/start-enterprise-gateway.sh && \
       touch /usr/local/share/jupyter/enterprise-gateway.log && \
       chmod 0666 /usr/local/share/jupyter/enterprise-gateway.log && \
       rm -f /usr/local/bin/bootstrap-kernel.sh


RUN curl -s https://archive.apache.org/dist/spark/spark-${SPARK_VER}/spark-${SPARK_VE
hadoop2.7.tgz | \
    tar -xz -C /opt && \
    ln -s ${SPARK_HOME}-${SPARK_VER}-bin-hadoop2.7 $SPARK_HOME && \
    mkdir -p /usr/hdp/current && \
    ln -s ${SPARK_HOME}-${SPARK_VER}-bin-hadoop2.7 /usr/hdp/current/spark2-client


CMD ["/usr/local/bin/start-enterprise-gateway.sh"]
EXPOSE 8888
WORKDIR /usr/local/bin
```

Para poder crear la imagen tenemos que ejecutar el siguiente comando en el directorio contenedor del fichero Dockerfile y de los archivos que se van a copiar en la imagen:

```
sudo docker build -t <usuario>/<nombre>:<tag> .
```

Si queremos usar la imagen Docker en otro ordenador y no queremos volver a crear la imagen, debemos tener la imagen en un repositorio como DockerHub. Para poder subir imágenes simplemente tenemos que usar la opción *docker push <imagen>* y para descargar *docker pull <imagen>.*

4.4.2 - Despliegue de Jupyter Enterprise Gateway en el clúster de Kubernetes

En entorno como Kubernetes debemos desplegar las aplicaciones vía ficheros YAML. Estos ficheros YAML declararán los recursos a crear por Kubernetes como pueden ser: PODs, servicios y deployments.

Lo primero que hacemos es crear un namespace donde se alojarán los distintos recursos relacionados con el funcionamiento de Jupyter Enterprise Gateway.

```
apiVersion: v1
kind: Namespace
metadata:
  name: enterprise-gateway
  labels:
    app: enterprise-gateway
```

De esta forma, todos los recursos que se creen dentro de este namespace van a poder ser localizados con una única instrucción (`kubectl get all –namespace enterprise-gateway`).

Kubernetes incluye un mecanismo de autenticación que permite administrar los permisos que puede realizar un usuario bajo ese rol. En nuestro caso, vamos a crear dos Roles que nos permitan controlar los permisos del POD que ejecutará Jupyter Enterprise Gateway los kernels Python.

```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRole
metadata:
  name: enterprise-gateway-controller
  labels:
    app: enterprise-gateway
    component: enterprise-gateway
rules:
  - apiGroups: [""]
    resources: ["pods", "namespaces", "services", "configmaps", "secrets",
"persistentvolumes", "persistentvolumeclaims"]
    verbs: ["get", "watch", "list", "create", "delete"]
  - apiGroups: ["rbac.authorization.k8s.io"]
    resources: ["rolebindings"]
    verbs: ["get", "list", "create", "delete"]
```

Este nuevo ClusterRole permite realizar las operaciones indicadas en *verbs* sobre los recursos que se encuentran en la lista *resources*.

Una vez configurados los permisos necesarios para la correcta interacción entre los diferentes PODs, creamos un recurso de tipo *Service*.

Este tipo de recurso es un conjunto de PODs que implementan una función específica. Además, los *Services* permiten exponer de forma interna o externa al clúster nuestra aplicación. Hay 3 tipos de servicios:

- **ClusterIP**: solo permite el acceso interno a la aplicación. Podemos acceder a la aplicación, pero esta no puede interactuar con el exterior. Es la opción por defecto. En la imagen podemos ver que, aunque WordPress haga uso de la MYSQL, el usuario no puede acceder.

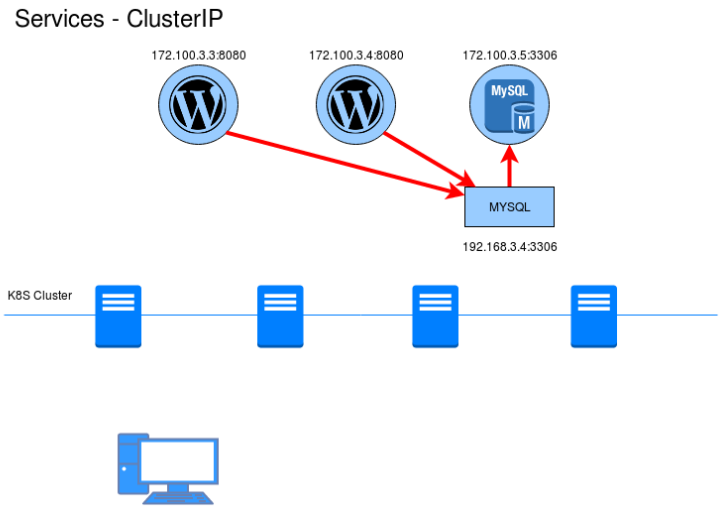


Ilustración 19 - Servicio tipo ClusterIP Fuente: https://www.josedomingo.org/

- **NodePort:** en cada nodo del clúster se abre un puerto aleatorio, siendo el mismo para todos. Podemos acceder a la aplicación usando la IP del nodo maestro y el puerto asignado. Como podemos ver en la imagen, el usuario accede al servicio via la IP del nodo maestro.

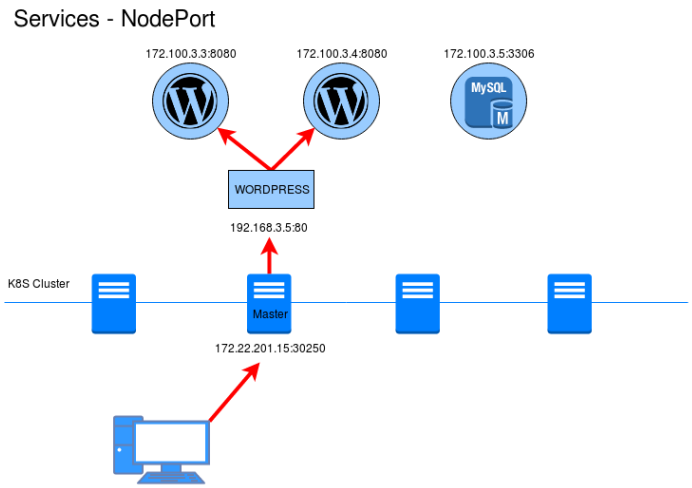


Ilustración 20 - Servicio tipo NodePort Fuente: https://www.josedomingo.org/

- **LoadBalancer:** este tipo de servicio únicamente está disponible en servicios de nube públicos (AWS, Google). El proveedor generará un balanceador de carga desde el que acceder. En la imagen podemos ver que, en función de la carga de los nodos, al acceder al servicio, puede atendernos la petición un nodo u otro.

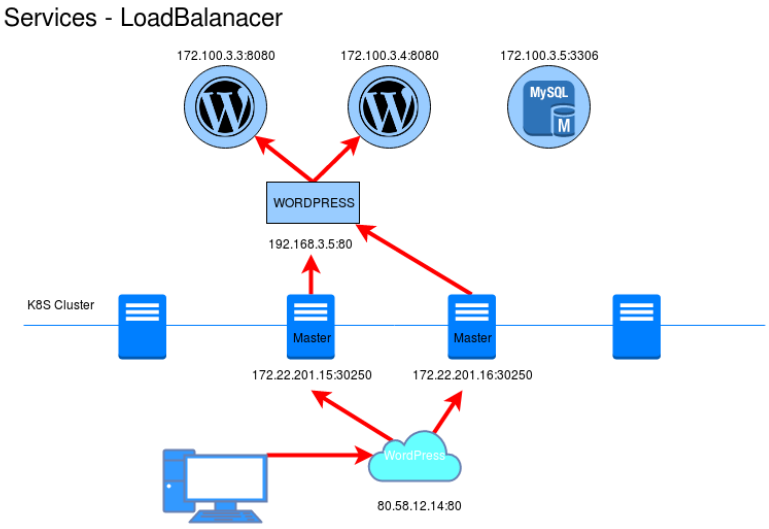


Ilustración 21- Servicio tipo LoadBalancer Fuente: https://www.josedomingo.org/

Durante la realización del proyecto se ha trabajado tanto con servicios NodePort como LoadBalancer. Al trabajar en la nube de Amazon, decidí usar LoadBalancer ya que permite automatizar el despliegue de forma más sencilla.

Dicho servicio, creado en el namespace enterprise-gateway, se expone al exterior en el puerto 8888 y es el proveedor quién se encarga de generar de forma automática la URL.

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: enterprise-gateway
    component: enterprise-gateway
  name: enterprise-gateway
  namespace: enterprise-gateway
  annotations:
    service.beta.kubernetes.io/aws-load-balancer-internal: 0.0.0.0/0
    service.beta.kubernetes.io/aws-load-balancer-connection-idle-
timeout: "600"
    service.beta.kubernetes.io/aws-load-balancer-additional-resource-
tags: "Name=CD4ML-ELB"

spec:
  ports:
  - name: http
    port: 8888
    targetPort: 8888
  selector:
    gateway-selector: enterprise-gateway
  type: LoadBalancer
```

Por último, faltaría especificar la aplicación que vamos a ejecutar en el clúster. Para ello existe el recurso *deployment*, quién se encarga de crear los PODs necesarios para ejecutar la aplicación desde una imagen Docker.

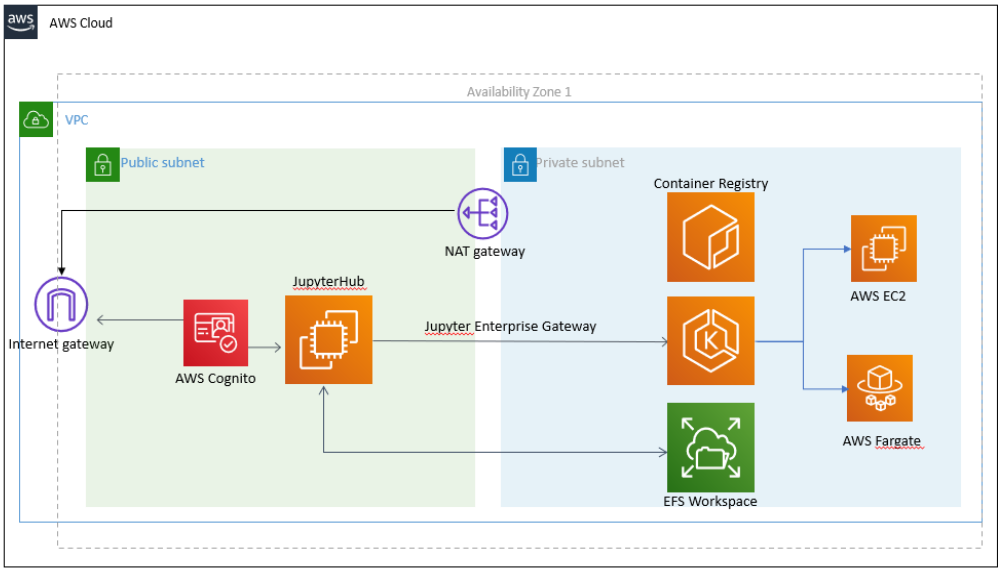
Las partes más importantes de este recurso son:

- **Spec.template.spec.env:** En esta sección se declaran las variables que va a tener el POD que va a ejecutar la aplicación. De esta forma, podemos pasarle variables a nuestra imagen Docker. Recordemos que durante el inicio de Jupyter Enterprise Gateway se monta una unidad EFS haciendo uso de una variable de entorno. Dicha variable viene declara aquí.
- **Image:** se especifica que imagen del repositorio va a ser usada
- **SecurityContext**: permite hacer uso el comando *mount* dentro de un POD

```
apiVersion: apps/v1
kind: Deployment
spec:
  replicas: 1
  template:
    metadata:
    spec:
      # Created above.
      serviceAccountName: enterprise-gateway-sa
      containers:
      - env:
        - name: EG_PORT
          value: "8888"

        - name: EG_NAMESPACE
          value: "enterprise-gateway"

        - name: EG_KERNEL_CLUSTER_ROLE
          value: "kernel-controller"

        - name: EG_KERNEL_LAUNCH_TIMEOUT
          value: "600"

        - name: EG_POSTGRES_IP
          value: "10.98.10.14"

        - name: KERNEL_EFS_ID
          value: "fs-671cf0ad.efs.eu-west-1.amazonaws.com"

        - name: EG_POSTGRES_PORT
          value: "5432"

        image: lucabem/tfg-jeg:dev8
        imagePullPolicy: IfNotPresent
        name: enterprise-gateway
        securityContext:
          capabilities:
            add: ["SYS_ADMIN"]
        ports:
        - containerPort: 8888
```

4.5 – Creación de la arquitectura en Amazon usando Terraform

La plataforma de data science, que estará alojada en Amazon Web Services, estará basada en la arquitectura mostrada anteriormente (ver Ilustración 11). A continuación, se muestra la misma imagen con el fin de facilitar su explicación:



4.5.1 - Configuración de la red

El componente principal sobre el que estarán alojados todos los componentes es la VPC, una red virtual en la nube que se dividirá en 3 subredes:

- Una subred pública donde se alojará la máquina de JupyterHub.
- Dos subredes privadas donde se alojará el clúster de Kubernetes, y, por tanto, todos los kernels que se estén ejecutando.

La VPC tendrá como opciones de configuración:

- Rango de IPS: 10.98.0.0/16.
- Permitimos el soporte y resolución de IPs a través de DNS. Activando dicha opción permitimos identificar nuestras máquinas haciendo uso del DNS.

Una vez creada la VPC tenemos que asociar las diferentes subredes. Cada una de las subredes tendrá asociadas:

- El **ID de la VPC** creada anteriormente.
- El **rango de bloques** para dicha subred: dicho rango debe estar contenido en el rango de la VPC (10.98.1.0/24, 10.98.2.0/24, 10.98.3.0/24)
- **Zona de disponibilidad**: dentro de cada región existen diferentes zonas de disponibilidad que evitan el colapso del sistema si se cae una zona. Por ello es recomendable replicar la arquitectura (o los elementos criticas) en diferentes zonas.
- **Asignar IP pública** durante la creación de una máquina: booleano que indica si se asigna una IP publica a una instancia alojada en dicha subred.

Uno de los requisitos necesarios para el correcto funcionamiento del clúster de Kubernetes es que este tenga acceso a Internet. Para poder permitir la salida a Internet desde una red privada, y que esta no sea accesible desde Internet, usamos las tablas de enrutamiento:

Las tablas de enrutamiento permiten dirigir en tráfico entre redes. En nuestro caso vamos a usar dos tablas de enrutamiento:

- Tabla de enrutamiento con salida a Internet.
- Tabla de enrutamiento privada.

La tabla de enrutamiento con salida a internet estará asociada a la subred pública y tendrá asociada una puerta de enlace o *gateway*. De esta forma, todas las máquinas, o también llamadas instancias, que se alojen en esta subred serán accesibles desde Internet (tendrán una IP pública) y podrás acceder a Internet desde ellas.

La tabla de enrutamiento privada tratará de redirigir las peticiones de acceso a Internet desde las subredes privadas a Internet a través de la subred pública. Para ello, creamos una NAT Gateway y la asociamos a la red pública.

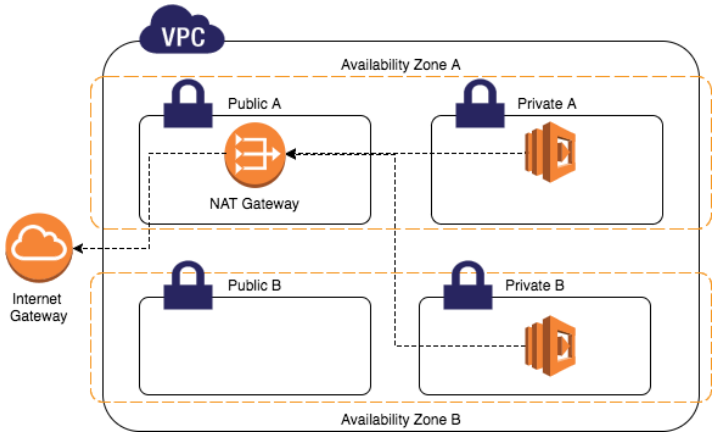


Ilustración 22 - Arquitectura VPC

4.5.2 - Creación del clúster de Kubernetes

AWS proporciona EKS, Elastic Kubernetes Service, un servicio gestionado de clúster de Kubernetes. Amazon se encarga de la gestión del propio clúster, por lo que este tipo de clústeres carece de un nodo maestro.

La creación de un EKS es bastante sencilla, ya que únicamente necesitamos tener una VPC cuyas subredes tengan acceso a internet. En el momento de seleccionar las subredes debemos tener en cuenta que será el lugar donde se alojarán los diferentes nodos esclavos.

Cada EKS puede contener diferentes grupos de nodos. Un grupo manejados de nodos gestiona el ciclo de vida de los nodos, pudiendo reducir o aumentar dicho número en función de la carga del sistema.

Cada grupo de nodos viene determinado por el tipo de instancia EC2 (o máquina), así como de una serie de parámetros que permiten gestionar el escalado del grupo:

- **Tamaño deseado**: número de instancias que se crean al iniciar el grupo de nodos.
- **Tamaño máximo**: número máximo de instancias que permite el grupo de nodos. Nunca habrá más instancias que dicho valor.
- **Tamaño mínimo**: número mínimo de instancias que permite el grupo de nodos. Nunca habrá menos instancias que dicho valor.

Por defecto, un grupo manejado de nodos no tiene activada la autogestión de las cargas, por lo que no se va a reducir/aumentar de forma automática el número de nodos.

4.5.3 - Configuración del sistema EFS

Amazon Elastic File System, o EFS, es un servicio que proporciona un sistema NFS totalmente administrado que permite almacenar grandes volúmenes de información en la nube de Amazon. Los sistemas EFS permiten el escalado automático de espacio sin necesidad de aprovisionar almacenamiento de forma estática.

El principal uso del EFS en nuestra plataforma es proporcionar una capa de persistencia de los datos. Dentro del EFS se guardarán todas las librerías, scripts de Python, entorno… generados por los usuarios.

Un sistema EFS está compuesto por diferentes puntos de montaje, o *mount points,* que permiten a las instancias montar dicho sistema NFS. Un requisito obligatorio es tener varios puntos de montaje en diferentes regiones con el fin de mantener la accesibilidad al EFS.

Cada uno de los puntos de montaje tendrá asignado una subred (debe pertenecer a la VPC del EFS), un grupo de seguridad y, como hemos comentado, una región.

El grupo de seguridad limita el rango de IPs que pueden montar el servicio. Dicho grupo debe tener una regla de entrada que permita el tráfico NFS (puerto 2049/TCP) dentro de la subred.

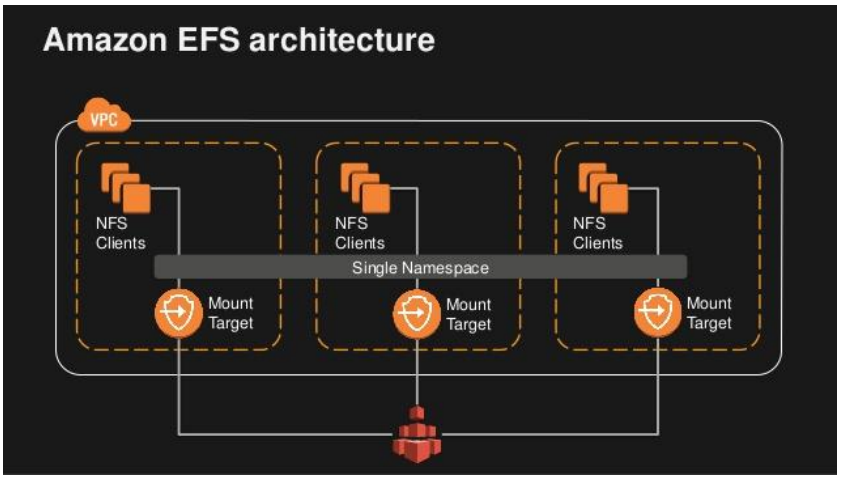


Ilustración 23 - Arquitectura EFS

Dicho NFS se montará, en tiempo de ejecución, en los PODs haciendo uso de las variables de entorno explicadas anteriormente.

4.5.4 - Configuración de la máquina de JupyterHub

La máquina de JupyterHub será el punto de unión entre los usuarios y el clúster de Kubernetes. Dicha máquina estará alojada en la subred pública y únicamente tendrá abiertos los puertos NFS y HTTPS, habilitados mediante grupos de seguridad.

La máquina tendrá, entre otros servicios:

- **JupyterHub**: servicio que permite acceder a las funcionalidades de Jupyter Notebook y Jupyter Enterprise Gateway.

- **PostgreSQL**: base de datos contra la que realizará consultas para obtener información del usuario y proyecto en el que está trabajando (recursos de CPU y memoria).

- **Directorio montando el EFS**: todos los usuarios tendrán su directorio Home en el sistema EFS, liberando espacio en la máquina EC2. Estará montado en la ruta */usr/local/share/jupyter/kernels*

Para evitar tener que configurar la máquina cada vez que se levante la plataforma, se ha creado un script de instalación y puesta a punto de los diferentes servicios explicados anteriormente.

Para añadir una capa extra de seguridad a Jupyterhub, haremos uso de AWS Cognito. Este servicio permite gestionar usuarios de forma rápida y sencilla de una forma externa. Entre otras cosas, AWS Cognito permite crear/eliminar usuarios de forma que todo aquel usuario que no aparece en Cognito no tendrá acceso JupyterHub. De esta forma, podemos dar de baja a un usuario mientras que mantendremos su directorio Home sin necesidad de borrarlo.

4.6 – Automatización de la plataforma vía Terraform

Uno de los principales problemas de la plataforma es su coste cuando no se está usando. La mayoría de los servicios de Amazon cobran por horas (salvo Fargate que cobra por carga), por lo que mantener activa la plataforma durante largos periodos de inactividad es muy ineficiente en términos económicos.

Terraform permite automatizar el proceso de creación y puesta en marcha de la arquitectura. Dicha tecnología trabaja mediante ficheros de configuración generados en una sintaxis propia semejante a JSON.

Cada uno de los recursos de Terraform se asemeja a un objeto en una programación orienta a objetos (POO), teniendo una serie de atributos con métodos *getter/setter* definidos. Por ejemplo, en el siguiente ejemplo creamos un recurso *aws_vpc*:

```
resource "aws_vpc" "CD4ML-VPC-Terraform" {
  cidr_block           = "10.98.0.0/16"
  enable_dns_hostnames = true
  enable_dns_support   = true
  instance_tenancy     = "default"

  tags = {
    "Name"                              = "CD4ML-VPC-Terraform"
    "Billing"                           = "CD4ML"
    "kubernetes.io/cluster/CD4ML-k8s" = "shared"
  }
}
```

Este recurso crea una red privada en la nube con un rango de IPs especificado y con una serie de opciones habilitadas. El bloque de *tags* es optativo. La VPC creada tendrá una serie de atributos a los que podemos acceder destre otros recursos como el id generado.

En el siguiente ejemplo vemos como se hacer referencia a un atributo de un recurso desde otro (si ambos están en el mismo módulo):

```
resource "aws_subnet" "CD4ML-Private-1-Terraform" {
  vpc_id                  = aws_vpc.CD4ML-VPC-Terraform.id
  cidr_block              = "10.98.11.0/24"
  availability_zone       = "eu-west-1a"
  map_public_ip_on_launch = false

  tags = {
    "Billing"                           = "CD4ML"
    "Name"                              = "CD4ML-Private-1-
Terraform"
    "kubernetes.io/cluster/CD4ML-k8s" = "shared"
  }
}
```

Un tag especial es *kubernetes.io/cluster/CD4ML-k8s.* Este tag informa al servicio EKS que dicha subred está disponible para levantar los nodos del clúster.

4.6.1 - Estructura de ficheros Terraform

La unidad mínima de Terraform es el módulo. Un módulo es un conjunto de ficheros con un objetivo común, permitiendo dar una estructura lógica a los ficheros de Terraform.

En nuestro caso, la estructura de módulos es la siguiente:

```
├── cognito
│   ├── output.tf
│   ├── user-pool.tf
│   └── variables.tf
├── efs
│   ├── efs.tf
│   ├── output.tf
│   └── variables.tf
├── eks
│   ├── eks-cluster.tf
│   ├── eks_fargate.tf
│   ├── eks_nodeGroup.tf
│   ├── policy_eks_cluster.tf
│   ├── policy_eks_nodeGroup.tf
│   ├── policy_fargate.tf
│   ├── sg_eks_node.tf
│   └── variables.tf
├── jupyterhub
│   ├── ec2-jupyterhub.tf
│   ├── output.tf
│   └── variables.tf
└── network
    ├── gateways.tf
    ├── output.tf
    ├── route-tables.tf
    ├── subnets.tf
    └── vpc.tf
```

La mayoría de los módulos tienen dos ficheros con el mismo nombre:

- ***Variables.tf:*** permite usar variables externas al módulo. Podemos importar variables/atributos de recursos generados en otros módulos, interconectando todos los módulos entre sí.

```
variable "subnet_private_1" {
  type        = string
  description = "ID de la subnet 1."
}

variable "subnet_private_2" {
  type        = string
  description = "ID de la subnet 2."
}
```

  El valor de las variables se especifica en el fichero *modules.tf*, dentro de la declaración de cada módulo

```
module "efs" {
  source = "./modules/efs"
  subnet_private_1        = "${module.network.subnet_private_1}"
  subnet_private_2        = "${module.network.subnet_private_2}"
  vpc_id                  = "${module.network.vpc_id}"
}
```

- ***Output.tf:*** Cada módulo puede producir una serie de atributos/valores de salida. Para poder acceder a dichos valores desde otro módulo, tenemos que declarar qué variables vamos a exportar y usar más tarde en un fichero *variables.tf* de otro módulo. Siguiendo el ejemplo anterior, las variables [*subnet_private_1, subnet_private_2, vpc_id*] vienen declaras en el fichero de salida:

```
output "vpc_id" {
  value = aws_vpc.CD4ML-VPC-Terraform.id
}

output "subnet_private_1" {
  value = aws_subnet.CD4ML-Private-1-Terraform.id
}

output "subnet_private_2" {
  value = aws_subnet.CD4ML-Private-2-Terraform.id
}

output "subnet_public_1" {
  value = aws_subnet.CD4ML-Public-1-Terraform.id
}
```

4.6.2 - Implementación de lógica en Terraform

Terraform es un lenguaje declarativo cuya lógica se realiza en tiempo de compilación, sin poder realizar condicionales en tiempo de ejecución.

El principal problema reside en la persistencia de los datos en el sistema EFS y en la máquina de JupyterHub. Cada vez que Terraform levanta la plataforma, se crean un nuevo EFS vacío y cuando se apaga, el contenido de este se pierde.

Para paliar el problema se ha trabajado en una solución que consta de dos objetivos:

1. Conseguir la **persistencia de los datos del EFS**: directorios Home del usuario, datasets, notebooks de trabajo, entornos…
2. **Persistencia en la máquina de JupyterHub**: ID de los usuarios para poder reasignar cada usuario con su directorio Home del EFS, librerías instaladas globalmente, base de datos…

4.6.2.1 - Persistencia del sistema EFS

El objetivo es realizar una copia de seguridad siempre que se destruya la plataforma (*Terraform destroy*) y se cargue dicha copia de seguridad cuando se levante la plataforma (salvo la primera vez).

En el módulo de EFS incluimos un recurso *null*, útil para realizar diferentes acciones sin interferir en el proceso, que se encarga de establecer un *gatillo*, o *trigger,* que se ejecute siempre.

Dicho recurso ejecuta un comando en local que se encarga de crear un trabajo de copia de seguridad dado el ARN del EFS. Esta variable está alojada en un fichero que contiene información sobre el EFS creado.

```
resource "null_resource" "CD4ML-BackUp-EFS" {
  triggers = {
    always_run = "${timestamp()}"
  }

  provisioner "local-exec" {
    when    = destroy
    command = "<comando para realizar backup>"
  }
}
```

El fichero resultado de este comando, *ficheros/punto_recuperacion.info,* servirá como condicional de existencia para próximos lanzamientos de la plataforma. La primera vez que lanzamos la plataforma no existe dicho fichero, por lo que Terraform es el encargado de crear el EFS.

En cambio, la existencia de ese fichero indica que el EFS tiene que ser recuperado desde una copia de seguridad. Para lograrlo, utilizamos otro recurso *null* que se encargue de recuperar el EFS. Además, dicho recursos se encargará de parar la ejecución de Terraform hasta que el nuevo EFS esté creado y así evitar errores de referencias. Por último, obtenemos información sobre el nuevo EFS creado a través de un script de Python que permite exportar el resultado como variable de Terraform.

4.6.2.2 - Persistencia de la máquina de JupyterHub

El principal objetivo de mantener una copia de la máquina de JupyterHub, y partir desde la última, se debe a la necesidad de mantener a los usuarios creados en dicha máquina.

Cada usuario tiene asignado un ID que le permite acceder a su sistema de ficheros (directorio Home). Si tuviéramos que crear cada vez los usuarios, en función del orden de creación podríamos tener inconsistencias en el directorio NFS.

USUARIO	1º ID JUPYTERHUB	OWNER HOME	2º ID JUPYTERHUB	OWNER HOME
LUIS	1001	LUIS	1002	PEDRO
PEDRO	1002	PEDRO	1001	LUIS

Si no tuviéramos una copia de la máquina, podríamos llegar a inconsistencias entre IDs y directorio HOME. Durante el primer día de la plataforma crearíamos dos usuarios: Luis y Pedro. Dichos usuarios serían propietarios de sus propios directorios, ya que el ID del propietario coincidiría con el suyo.

En cambio, el segundo día de levantar la plataforma podríamos crear primero el usuario Pedro y luego Luis. Esto provocaría una permuta en los IDs, asignando al usuario Luis el directorio HOME de Pedro y viceversa.

Para evitarlo, cada vez que se apague la plataforma (vía Terraform) se realiza una instantánea de la máquina y guardamos su ID en un fichero. La próxima vez, Terraform consultará dicho fichero y tratará de partir desde esa instantánea. Si no existe dicho fichero, cargará una máquina base si ningún usuario (por defecto).

5 – Seguimiento y control

A pesar de realizar una exhaustiva planificación del proyecto, el uso de tecnologías punteras junto a una estimación optimista en ciertos campos ha producido un pequeño desvío en las horas programadas. En esta sección se detallan los motivos más importantes que han producido dicha desviación.

El número de horas planificadas para la realización del proyecto eran 300 horas, aunque se ha producido una desviación del 15% respecto a la planificación, dedicando un total de 350 horas aproximadamente.

Las horas planificadas en el diagrama de Gantt son las siguientes:

- Planificación: 20 horas
- Análisis: 25 horas
- Diseño: 50 horas
- Implementación: 170 horas
- Memoria: 15 horas
- Seguimiento y control: 20

Las 50 horas de desviación se han producido en el apartado *Implementación*. Principalmente se ha producido en la fase de *Automatizar despliegue* debido al desconocimiento de la tecnología Terraform.

Como se ha comentado durante el apartado *Automatización de la plataforma vía Terraform,* esta tecnología carece de una fuerte lógica implementada. Esto impedía realizar comprobaciones de carácter dinámico durante la ejecución del código, generando la necesidad de implementarla por mí mismo.

Como resultado de este retraso, no se ha podido completar algunos extras que se tenían pensados como la creación de un repositorio de imágenes dentro de la nube de Amazon o la persistencia del sistema de gestión de usuarios de AWS Cognito.

5.1 - Situación durante COVID19

Durante el período de alarma no ha existido ningún problema y/o retraso provocado por la situación de alarma que hemos vivido. Durante este período he seguido realizando el proyecto manteniendo una reunión semanal con compañeros de la empresa.

Asimismo, se ha mantenido la comunicación con César, tutor del proyecto, a través de medios telemáticos como el correo.

6 – Conclusiones

Durante esta sección se exponen una serie de conclusiones desde el lado técnico de la plataforma (resultados, gestión) como resultados académicos (nivel de autonomía, aprendizaje).

6.1 – Conclusiones técnicas

En la última semana del proyecto se realizó una pequeña prueba de estrés durante 8 horas. Durante esas horas, los data science, quienes estuvieron generando modelos matemáticos, mostraron un alto grado de conformidad con la solución propuesta.

Está prueba acabó sin errores y con una monitorización continua a través de Prometheus, un servidor que permite monitorizar diferentes métricas relacionadas con el cluster de Kubernetes.

6.1 – Conclusiones académicas

Este proyecto me ha permitido trabajar con tecnologías punteras como Terraform y AWS, además de mejorar aspectos de programación concurrente. A su vez, me ha permitido realizar una de mis primeras colaboraciones en un proyecto de *open source* como Jupyter Enterprise Gateway.

Al mismo tiempo, me ha permitido envolverme en un proyecto real junto a un grupo multidisciplinar de expertos en temas de Big Data.

En definitiva, este proyecto me ha hecho madurar los conocimientos adquiridos durante la carrera al tener un elevado grado de autonomía y decisión, poniendo en práctica mucho de lo aprendido en asignaturas como: diseño de base de datos, sistemas operativos, redes y seguridad.

7 – Bibliografía

Durante esta sección se presenta la bibliografía utilizada durante la realización del proyecto.

- Kubernetes
  - *Setting up a multi-node Kubernetes cluster on a laptop - https://medium.com/@genekuo/setting-up-a-multi-node-kubernetes-cluster-on-a-laptop-69ae3e3d0f7c*
  - *Documentación oficial - https://kubernetes.io/*
  - *Tipos de servicios en Kubernetes - https://www.josedomingo.org/*

- AWS
  - *Documentación oficial - https://docs.aws.amazon.com/*

- Jupyter Enterprise Gateway
  - *Documentación oficial - https://jupyter-enterprise-gateway.readthedocs.io*
  - *Jupyter Enterprise Gateway - https://github.com/jupyter/enterprise_gateway*

- JupyterHub
  - *Documentación oficial - https://jupyterhub.readthedocs.io/en/stable/*
  - *JupyterHub - https://github.com/jupyterhub/jupyterhub*

- Jupyter Notebook
  - *Documentación oficial – https://jupyter.org/*
  - *Jupyter Notebook – https://github.com/jupyter/notebook*

- Docker
  - *Documentación oficial - https://docs.docker.com/*

- Terraform
  - *Documentación oficial – https://www.terraform.io/*
  - *Terraform - https://github.com/hashicorp/terraform*